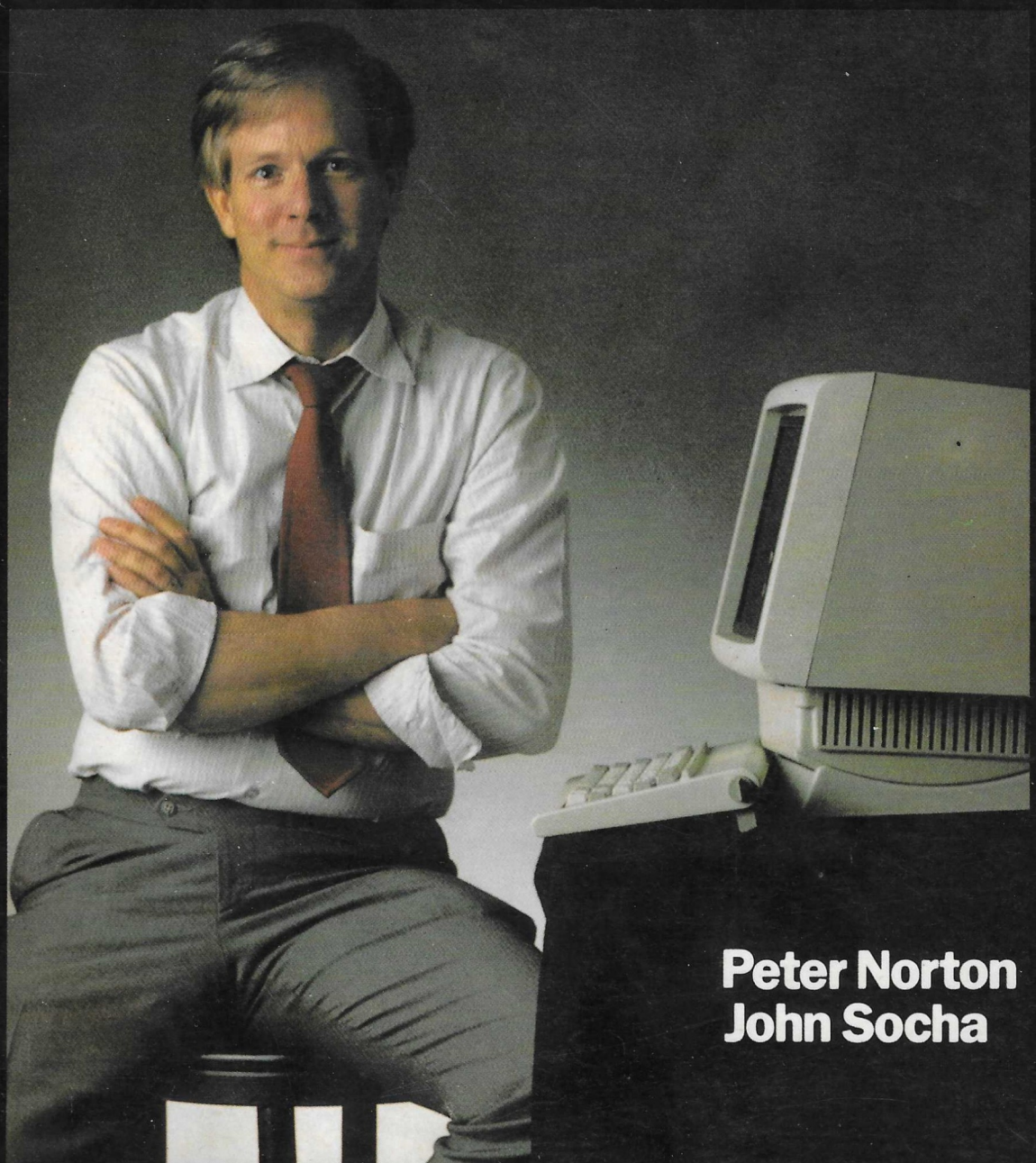
PETER NORTON
LINGUAGEM ASSEMBLY PARA IBM PC



Peter Norton
John Socha

Editora Campus

PETER NORTON
LINGUAGEM ASSEMBLY PARA
IBM PC

Peter Norton

O autor de *Desvendando o* **PC** e **PC-DOS** *Como Usar o DOS com Inteligência.*

John Socha

da Peter Norton Computing.

Tradução
Daniel Vieira

1ª EDIÇÃO

1ª REIMPRESSÃO

Editora Campus Ltda.

Uma Editora da Elsevier, um nome com 400 anos de tradição.

Do original:
Peter Norton's Assembly Language Book for the IBM PC

Copyright © 1986 by Brady Books, a division of Simon & Schuster, Inc.
All rights reserved.

© 1988, Editora Campus Ltda. 1ª Reimpressão, 1989.

Todos os direitos reservados e protegidos pela Lei 5988 de 14/12/73.
Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.
Todo o esforço foi feito para fornecer a mais completa e adequada informação. Contudo a editora e o(s) autor(es) não assumem responsabilidade pelos resultados e uso da informação fornecida. Recomendamos aos leitores testar a informação antes de sua efetiva utilização.

Capa
Otavio Studart

Projeto Gráfico, Composição e Revisão
Editora Campus Ltda.
Qualidade internacional a serviço do autor e do leitor nacional.
Rua Barão de Itapagipe 55 Rio Comprido
Telefone: (021) 293-6443 Telex: (021) 32606 EDCP BR
20261 Rio de Janeiro RJ Brasil
Endereço Telegráfico: CAMPUSRIO

ISBN 85-7001-511-9
(Edição original: ISBN 0-13-661901-0 Brady Book Published by Prentice Hall Press, New York, New York.)

Ficha Catalográfica
CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ.

N778p

Norton, Peter, 1943-
Peter Norton, linguagem ASSEMBLY para IBM PC / Peter Norton e John Socha; tradução de Daniel Vieira. — Rio de Janeiro: Campus, 1988.

Tradução de: Peter Norton's ASSEMBLY Language Book for the IBM PC.
Inclui apêndices
ISBN 85-7001-511-9

1. ASSEMBLER (Linguagem de programação de computador). 2. Microcomputadores — Programação. I. Socha, John, 1958- II. Título.

87-1072

CDD — 001.6424

INTRODUÇÃO

Quando você terminar de ler este livro, saberá como escrever programas completos em linguagem assembly: editores de texto, utilitários etc. No meio do caminho, você também vai aprender muitas técnicas que os programadores profissionais usam para tornar o seu trabalho mais simples. Estas técnicas, incluindo projeto modular e refinamentos sucessivos, duplicarão ou triplicarão a sua velocidade de programação, bem como o ajudarão a escrever programas mais legíveis e confiáveis.

A técnica de refinamentos sucessivos, em particular, alivia muito o trabalho de escrita em programas complexos. Se você já teve esse sentimento de "por-onde-eu-começo", então verá que o refinamento sucessivo pode dar um modo simples e natural de se escrever programas. E também é divertido!

No entanto, este livro não é só teoria. Nós montaremos um programa também. O programa é chamado Dskpatch (de Disk Patch), e ele será útil a você por diversas razões. Primeiro de tudo, você verá o refinamento sucessivo e o projeto modular em um programa real, de modo que terá uma oportunidade de observar por que essas técnicas são tão úteis. Além disso, como você verá logo em seguida, o Dskpatch é, por direito próprio, um editor de tela completa para uso geral em setores de disco — um programa que você poderá usar, em todo ou em parte, muito após o término deste livro.

Por Que Linguagem Assembly?

Iremos assumir que você está lendo este livro porque deseja aprender linguagem assembly. Mas você pode não estar bem certo do motivo para aprendê-la.

Uma razão, talvez a menos óbvia, é que os programas em linguagem assembly sejam o coração de qualquer IBM PC ou computador compatível. Em relação a todas as outras linguagens de programação, a linguagem assembly é o denominador comum entre elas. Ela leva você mais perto da máquina do que as linguagens de alto nível, de modo que o aprendizado da linguagem assembly também significa aprender a entender o microprocessador 8088 dentro do seu computador. Ensinar-lhe-emos as instruções do microprocessador 8088, como os autores dos livros de introdução, mas iremos bem mais adiante e também veremos certo material avançado que será de inestimável valor quando você começar a escrever seus próprios programas.

Uma vez você entendendo o microprocessador 8088 dentro do seu IBM PC, muitos elementos que você verá em outros programas e em linguagens de alto nível terão maior significado. Por exemplo, você pode ter notado que o maior inteiro permitido no BASIC é 32767. De onde vem esse número? Esse é um número estranho para um limite máximo. Mas, como você verá adiante, o número 32767 é diretamente relacionado ao modo com que o IBM PC armazena os números.

Depois, você também pode estar interessado em velocidade ou tamanho. Via de regra, os programas em linguagem assembly são muito mais rápidos do que os escritos em qualquer outra linguagem. Programas típicos em linguagens assembly são duas ou três vezes mais rápidos do que os programas equivalentes em C ou Pascal, e geralmente ultrapassam os programas em BASIC interpretado em 15 vezes ou mais. Os programas em linguagem assembly também são menores. O programa Dskpatch que montaremos neste livro ocupa cerca de um kbyte. Comparado com os programas em geral, isto é muito pouco. Um programa semelhante escrito em C ou Pascal ocuparia cerca de dez vezes mais espaço. Por estas razões, entre outras, a "Lotus Development Corporation" escreveu o 1-2-3 inteiramente em linguagem assembly.

Os programas em linguagem assembly também possibilitam um total acesso a todas as características do seu computador. Uma série de programas, como o SideKick, ProKey e SuperKey, permanecem na memória após a sua execução. Tais programas mudam o modo de trabalho da máquina, e usam características do sistema disponíveis somente para programas em linguagem assembly.

Dskpatch

Em nosso trabalho com a linguagem assembly, veremos diretamente os setores de disco, apresentando caracteres e números armazenados lá pelo DOS em notação hexadecimal. Dskpatch é um editor de tela completa para discos, e nos permitirá alterar caracteres e números em um setor do disco. Usando Dskpatch você poderá, por exemplo, ver os setores onde o DOS armazena o diretório de um disco e alterar os nomes dos arquivos ou outras informações. Esse é um bom modo de se aprender como o DOS armazena informações no disco.

Com Dskpatch, no entanto, você ganhará mais do que um programa. Ele contém cerca de 50 sub-rotinas. Muitas delas são sub-rotinas de uso geral que você poderá utilizar para incluir em seus próprios programas. Assim, este livro não somente é uma introdução ao 8088 e à programação em linguagem assembly, mas também uma fonte de sub-rotinas úteis.

Além disso, um editor de tela completa precisa usar características específicas à família IBM PC de computadores. Através dos exemplos neste livro, você também aprenderá a escrever programas úteis para IBM PCs, ATs, ou computadores compatíveis, como o COMPAQ.

Requisitos de Equipamento

Que equipamento você precisará para rodar os exemplos deste livro? Precisará de um IBM PC ou compatível com pelo menos 128K de memória e um drive de disco. Você também irá precisar da versão 2.00 ou posterior do PC-DOS (ou MS-DOS). E, a partir da Parte II, precisará do Macro Assembler da IBM ou da Microsoft.

Organização Deste Livro

Este livro está dividido em três partes, cada qual com uma ênfase diferente. Sabendo ou não alguma coisa sobre microprocessadores ou linguagem assembly, você verá diversas seções de interesse.

A Parte I focaliza o microprocessador 8088. Aqui, você aprenderá os mistérios dos bits, bytes e linguagem de máquina. Cada um dos sete capítulos contém uma riqueza enorme em exemplos reais que utilizam um programa chamado Debug, que vem com o disco do DOS. O Debug permitirá que você veja dentro do famoso microprocessador 8088 embutido dentro do seu PC enquanto está trabalhando sob o DOS. A Parte I assume somente que você possui um conhecimento elementar de BASIC e sabe como trabalhar com o seu computador.

A Parte II, Capítulos de 8 a 16, passa para a linguagem assembly, e como escrever programas no assembler. O método é progressivo, e ao invés de cobrirmos todos os detalhes da linguagem assembly em sim, iremos nos concentrar num grupo de comandos assembler que precisamos saber para escrever programas úteis.

Usaremos o assembler para reescrever alguns dos programas da Parte I, e depois partiremos para a criação do Dskpatch. Montaremos este programa lentamente, para que você aprenda a utilizar o método de refinamentos sucessivos para criar grandes programas. Também veremos técnicas como o projeto modular, que ajudam a escrever programas limpos. Como dissemos, estas técni-

cas simplificam a programação removendo algumas das complexidades normalmente associadas à escrita de programas em linguagem assembly.

Na Parte III, que se compõe dos Capítulos de 17 a 29, iremos nos deter no uso de características mais avançadas encontradas nos IBM PCs. Estas características incluem o movimento do cursor e o apagamento da tela.

Na Parte III também veremos as técnicas para depurar programas maiores em linguagem assembly. Os programas em linguagem assembly crescem muito rapidamente e podem facilmente ter duas ou mais páginas sem fazerem muita coisa (Dskpatch será maior). Embora estejamos usando essas técnicas de depuração em programas maiores do que algumas páginas, você verá que elas também são úteis com programas pequenos.

Dito isso, sem mais demora, vamos passar para o 8088 e ver como ele armazena os números.

SUMÁRIO

PARTE I
LINGUAGEM DE MÁQUINA 1

CAPÍTULO 1
DEBUG E ARITMÉTICA 3
Números Hexadecimais 4
Debug 4
Aritmética Hexa 5
Convertendo de Hexa para Decimal 6
Números Hexa de Cinco Dígitos 9
Convertendo de Decimal para Hexa 9
Números Negativos 11
Bits, Bytes, Palavras e Notação Binária 12
Complemento de Dois — Um Tipo Estranho de Número Negativo 14
Resumo 15

CAPÍTULO 2
ARITMÉTICA 8088 17
Registradores Como Variáveis 18
A Memória e o 8088 19
Adição ao Modo 8088 20
Subtração ao Modo 8088 22
Números Negativos no 8088 23
Bytes no 8088 23
Multiplicação e Divisão ao Modo 8088 24
Resumo 26

CAPÍTULO 3
IMPRIMINDO CARACTERES 27
INT — A Poderosa Interrupção 28
Uma Boa Saída — INT 20h 29
Um Programa de Duas Linhas — Juntando as Partes 30
Entrando Programas 30
MOVendo Dados Para os Registradores 31
Imprimindo um String de Caracteres 33
Resumo 34

CAPÍTULO 4
IMPRIMINDO NÚMEROS BINÁRIOS 35
Rotações e o Flag de Carry 36
Somando com o Flag de Carry 37
Loops 37
Imprimindo um Número Binário 39
O Comando de Prosseguimento 40
Resumo 40

CAPÍTULO 5
IMPRIMINDO EM HEXA **43**
Bits de Comparação e Status **44**
Imprimindo um Único Dígito Hexa **46**
Outra Instrução de Rotação **47**
Lógica e AND **49**
Juntando as Partes **50**
Resumo **51**

CAPÍTULO 6
LENDO CARACTERES **53**
Lendo um Caractere **54**
Lendo um Número Hexa de Único Dígito **54**
Lendo um Número Hexa de Dois Dígitos **55**
Resumo **56**

CAPÍTULO 7
PROCEDURES — PRIMAS DAS SUB-ROTINAS **57**
Procedures **58**
A Pilha e os Endereços de Retorno **59**
PUSH e POP **61**
Lendo Números Hexa com Mais Facilidade **61**
Resumo **63**

PARTE II
LINGUAGEM ASSEMBLY **65**

CAPÍTULO 8
BEM-VINDO AO ASSEMBLER **67**
Um Programa Sem Debug **68**
Criando Arquivos Fonte **70**
Linkeditando **70**
De Volta ao Debug **71**
Comentários **72**
Rótulos **72**
Resumo **74**

CAPÍTULO 9
PROCEDURES E O ASSEMBLER **75**
As Procedures do Assembler **76**
As Procedures de Saída em Hexa **78**
Rudimentos do Projeto Modular **80**
Um Esqueleto de Programa **81**
Resumo **81**

CAPÍTULO 10
IMPRIMINDO EM DECIMAL **83**
Relembrando a Conversão **84**
Alguns Truques **86**
O Funcionamento Interno **87**
Resumo **88**

CAPÍTULO 11
SEGMENTOS **89**
Seccionando a Memória do 8088 ..... **90**
Pseudo-Ops de Segmento **94**
A Pseudo-Op ASSUME **95**
CALLs Próximos e Distantes **96**
Mais Sobre a Instrução INT **98**
Vetores de Interrupção **99**
Resumo **99**

CAPÍTULO 12
ACERTANDO O CURSO **101**
Disquetes, Setores e Dskpatch **102**
O Plano do Jogo **104**
Resumo **104**

CAPÍTULO 13
PROJETO MODULAR **105**
Montagem Separada **106**
As Três Leis do Projeto Modular **108**
Resumo **111**

CAPÍTULO 14
COPIANDO A MEMÓRIA **113**
Modos de Endereçamento **114**
Adicionando Caracteres à Cópia **118**
Copiando 256 Bytes da Memória **120**
Resumo **123**

CAPÍTULO 15
COPIANDO UM SETOR DO DISCO **125**
Tornando a Vida Mais Fácil **126**
Formato do Arquivo Make **126**
Alterando Disp_set **127**
Lendo um Setor **128**
Resumo **131**

CAPÍTULO 16
MELHORANDO A APRESENTAÇÃO DO SETOR **133**
Adicionando Caracteres Gráficos **134**
Adicionando Endereços **136**
Adicionando Linhas Horizontais **138**
Adicionando Números **141**
Resumo **143**

PARTE III
O ROM BIOS DO IBM PC **145**

CAPÍTULO 17
AS ROTINAS DO ROM BIOS **147**
VIDEO_IO, As Rotinas do Rom Bios **148**

Movendo o Cursor 152
Ampliando o Uso de Variáveis 153
Imprimindo o Cabeçalho 156
Resumo 157

CAPÍTULO 18
O IMP_CAR DEFINITIVO 159
Um Novo IMP_CAR 160
Limpando Até o Fim da Linha 162
Resumo 164

CAPÍTULO 19
O CONTROLADOR 165
O Controlador 166
Lendo Outros Setores 170
Filosofia dos Capítulos Seguintes 172

CAPÍTULO 20
UM DESAFIO DE PROGRAMAÇÃO 173
Os Cursores Fantasma 174
Edição Simples 175
Outros Acréscimos e Mudanças em Dskpatch 176

CAPÍTULO 21
OS CURSORES FANTASMA 177
Os Cursores Fantasma 178
Alterando Atributos de Caracteres 182
Resumo 183

CAPÍTULO 22
EDIÇÃO SIMPLES 185
Movendo os Cursores Fantasma 186
Edição Simples 188
Resumo 190

CAPÍTULO 23
ENTRADA HEXA E DECIMAL 193
Entrada Hexa 194
Entrada Decimal 199
Resumo 201

CAPÍTULO 24
ENTRADA DO TECLADO MELHORADA 203
Um Novo LE_STRING 204

CAPÍTULO 25
À PROCURA DE ERROS 209
Consertando CONTROLA 210
Resumo 211

CAPÍTULO 26
GRAVANDO SETORES MODIFICADOS 213
Gravando no Disco 214

Mais Técnicas de Depuração 215
Fazendo um Mapa de Estrada 216
Descobrindo Erros 218
Symdeb 219
Depuração Simbólica 219
Troca de Tela 219
Resumo 221

CAPÍTULO 27
O OUTRO MEIO SETOR 223
Rolando Meio Setor 224
Resumo 226

PARTE IV
DIVERSOS 227

CAPÍTULO 28
RELOCAÇÃO 229
Segmentos Múltiplos 230
Relocação 233
Programas .COM versus .EXE 235

CAPÍTULO 29
MAIS SOBRE SEGMENTOS E ASSUME 237
Superposição de Segmentos 238
Mais Sobre ASSUME 239
Erros de Fase 240
Palavras Finais 240

APÊNDICE A
LISTAGEM DO DSKPATCH 243
Descrição das Rotinas 244
Listagens de Programas Para as Rotinas do Dskpatch 247

APÊNDICE B
ORDEM DE CARGA DOS SEGMENTOS 277
Ordem de Carga dos Segmentos 278
Erros de Fase 279
Arquivo EXE2BIN Não Pode Ser Convertido 281

APÊNDICE C
MENSAGENS DE ERRO COMUNS 283
MASM 284
LINK 284
EXE2BIN 285

APÊNDICE D
DIVERSAS TABELAS 287
Códigos de Caracteres ASCII 288
Códigos Estendidos do Teclado 290
Os Modos de Endereçamento 291
Funções de INT 10h 292
Funções de INT 21h 294

ÍNDICE REMISSIVO 295

PARTE I

LINGUAGEM DE MÁQUINA

1

DEBUG E ARITMÉTICA

Números Decimais

Debug

Aritmética Hexa

Convertendo de Hexa Para Decimal

Números Hexa de Cinco Dígitos

Convertendo de Decimal Para Hexa

Números Negativos

Bits, Bytes, Palavras e Notação Binária

Complemento de Dois — Um Tipo Estranho de Número Negativo

Resumo

Vamos começar nossa jornada pela linguagem, como os computadores contam. Isso pode parecer muito simples. Afinal de contas, contamos até 11 começando em um e aumentando passo a passo: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

Mas um computador não conta assim. Ao invés disso, ele conta até cinco da seguinte maneira: 1, 10, 11, 100, 101. Os números 10, 11, 100, e assim por diante, são números binários, baseados em um sistema de numeração com apenas dois dígitos, um e zero, ao invés dos dez associados ao nosso sistema decimal mais familiar. Assim sendo, o número binário é equivalente ao número decimal que conhecemos como dois.

Os números binários nos interessam porque eles são a forma usada pelo microprocessador 8088 dentro do seu IBM PC. Mas, embora os computadores trabalhem somente com séries de zeros e uns, esses números são muito difíceis de escrever. A solução? Números hexadecimais — uma forma mais compacta de se escrever os números binários. Neste capítulo, aprenderemos os dois modos de se escrever os números: hexadecimal e binário. E enquanto você aprende sobre a contagem por computadores, você também aprenderá como eles armazenam os números — em bits, bytes e palavras.

Se você já sabe sobre números binários e hexadecimais, bits, bytes e palavras, você pode saltar para o resumo do capítulo.

Números Hexadecimais

Como os números hexadecimais são mais fáceis de se lidar do que os números binários — pelo menos em termos de tamanho —, começaremos com o hexadecimal (hexa, para abreviar), e usaremos o DEBUG.COM, um programa que você encontrará em seu disco do PC/MS DOS. Usaremos o Debug aqui e nos capítulos seguintes para entrar e executar programas em linguagem de máquina em uma instrução de cada vez. Assim como o BASIC, o Debug fornece um ambiente interativo. Mas diferente do BASIC, ele não conhece os números decimais. Para o Debug, o número 10 é um número hexa — e não dez. E como o Debug só fala em hexadecimal, você precisa aprender algo sobre os números hexa. Mas primeiro vamos sair um pouco da rota e ver algo sobre o próprio Debug.

Debug

Por que esse programa tem o nome de Debug? Os *bugs*, no mundo de computador, são erros nos programas. Um programa de trabalho não possui bugs, enquanto um programa em desenvolvimento possui pelo menos um. Usando o Debug para executar um programa uma instrução de cada vez, e verificando como o programa funciona, podemos encontrar e corrigir os erros. Isto é conhecido como *debugging* (ou depuração), e daí o nome Debug.

De acordo com o folclore de computador, o termo debugging vem dos primórdios da computação e em particular, um dia em que o computador Mark I, em Harvard, teve uma falha. Após uma longa busca, os técnicos descobriram a fonte de seus problemas: uma pequena mariposa presa entre os contatos de um relé. Os técnicos removeram o inseto (*bug*) e escreveram uma nota no diário sobre o *debugging* do Mark I.

Encontre o Debug em seu disco do DOS e vamos começar. Você pode utilizar um disco de trabalho com uma cópia do Debug nele. O arquivo se chama DEBUG.COM, e iremos utilizá-lo bastante na Parte I deste livro.

Nota: Daqui em diante, em sessões interativas como esta, o texto que você digita estará enfatizado dentro da linha para distingui-lo das respostas do computador. Digite o texto, pressione a tecla Enter, e você deverá ver uma resposta igual às apresentadas nesta seção. Nem sempre as respostas serão as mesmas, pois seu computador provavelmente terá uma quantidade de memória diferente do computador em que escrevi este livro. (Começaremos a ver tais diferenças no próximo capítulo.) Além disso, note que usamos letras maiúsculas em todos os exemplos. Isto é somente para evitar qualquer confusão entre a letra maiúscula I (ele) e o número 1 (um). Se você preferir, poderá digitar todos os exemplos em letras minúsculas.

Agora, com estas poucas convenções em mente, inicie o Debug digitando seu nome após o pronto do DOS (que é A> neste exemplo):

```
A>DEBUG
-
```

O hífen que você vê em resposta ao seu comando é o símbolo de pronto do Debug, assim como A> é o pronto do DOS. Ele significa que o Debug está esperando por um comando.

Para deixar o Debug e retornar ao DOS, digite somente Q (*Quit* — encerrar) após o hífen do pronto e pressione Enter. Tente encerrar agora, se quiser, e depois retorne ao Debug:

```
-Q
A>DEBUG
-
```

Agora, podemos continuar aprendendo sobre os números hexadecimais.

Aritmética Hexa

Usaremos um comando do Debug chamado H. H é a abreviação de *Hexaritmética* e, como o nome sugere, ele soma e subtrai dois números hexa. Vejamos como H trabalha iniciando com 2 + 3. Sabemos que 2 + 3 = 5 para números decimais. Isto também é verdade para números hexadecimais? Assegure-se de ainda estar no Debug e, no hífen de pronto, digite o seguinte texto:

```
-H 3 2
0005 0001
```

O Debug imprime tanto a soma (0005) quanto a diferença (0001) entre 3 e 2. O comando de hexaritmética sempre calcula a soma e a diferença entre dois números, como fez aqui. E até aqui os resultados são os mesmos para números decimais e hexadecimais: 5 é a soma de 3 + 2 em decimal, e 1 é a sua diferença (3 – 2). Mas às vezes você pode encontrar algumas surpresas.

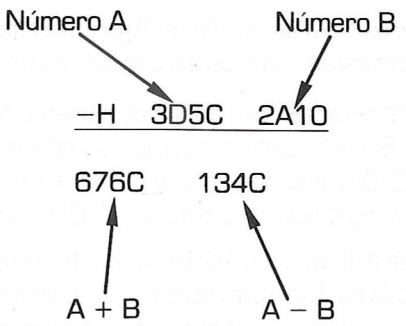


Figura 1-1. ***O Comando Hexaritmético.***

Por exemplo, e se digitarmos *H 2 3*, para somar e subtrair dois e três, ao invés de três e dois? Se tentarmos:

```
-H 2 3
0005 FFFF
```

obteremos FFFF ao invés de −1, para 2 − 3. Por mais estranho que pareça, no entanto, FFFF é um número. De fato, ele é o número −1 em hexadecimal.

Voltaremos a este −1 incomum brevemente. Mas primeiro vamos explorar o mundo dos números um pouco maiores para ver como um F pode aparecer em um número.

Para ver o que o comando Hexaritmético faz com números maiores, vamos tentar nove mais um, que nos daria o número decimal 10:

```
-H 9 1
000A 0008
```

Nove mais um é igual a A? Correto: A é o número hexa correspondente a dez. Agora, e se tentarmos para um número ainda maior, como 15?

```
-H 9 6
000F 0003
```

Se você tentar outros números entre dez e quinze, você terá 16 dígitos completos — de 0 a F (de 0 a 9, e de A a F). O nome hexadecimal vem de hexa (6) mais deca (10) que, quando combinados, representam 16. Os dígitos de 0 a 9 são os mesmos em hexadecimal e decimal; os dígitos hexa de A até F são equivalentes aos decimais de 10 a 15.

Por que o Debug fala em hexadecimal? Logo você verá que podemos escrever 256 valores diferentes com dois dígitos hexa. Como você já pode ter notado, 256 possui alguma relação com a unidade conhecida como byte, e o byte desempenha um papel muito importante nos computadores e neste livro. Você descobrirá mais sobre os bytes no final deste capítulo, mas por enquanto vamos nos concentrar em aprender sobre hexadecimais, o único sistema numérico conhecido pelo Debug, e a matemática hexa.

Convertendo Hexadecimal em Decimal

Até aqui vimos os números hexa de um dígito. Agora, vejamos como representar números hexa maiores, e como converter estes números para números decimais.

Assim como nos números decimais, montamos os números hexa de múltiplos dígitos adicionando mais dígitos à esquerda. Suponha, por exemplo, que somamos o número 1 ao maior número de um dígito no sistema decimal, 9. O resultado é um número de dois dígitos, 10 (dez). O que acontece quando somamos 1 ao maior número hexa de um dígito, F? Obtemos 10 novamente.

Mas espere, dez em hexa é na verdade 16, e não dez. Isto pode estar ficando um tanto confuso. Precisamos de algum modo de distinguir estes dois dez, de modo que daqui para frente colocaremos a letra h após qualquer número hexa. Assim, nós saberemos que 10h é 16 em hexadecimal,
 e 10 é o decimal 10.

Decimal	Dígito Hexa
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Figura 1-2. ***Números Hexadecimais.***

Agora chegamos ao ponto de como converter números entre hexa e decimal. Sabemos que 10h é 16, mas como convertemos um número hexa maior, como D3h, para um número decimal sem termos que contar até D3h a partir de 10h? Ou então, como converter o número decimal 173 para hexa?

Não podemos contar com a ajuda do Debug, pois ele não sabe nada de decimal. No Capítulo 10, iremos escrever um programa para converter um número hexa para a notação decimal, para que nossos programas possam falar conosco em decimal. Mas agora, teremos que fazer estas conversões a mão. Vamos começar retornando ao nosso conhecido mundo decimal.

O que significa o número 276? No colégio, aprendemos que 276 significa que temos duas centenas, sete dezenas, e seis unidades. Ou, esquematicamente:

```
2  x 100 = 200
 7 x  10 =  70
 6 x   1 =   6
-------------
276      = 276
```

Bem, isto certamente nos ajuda a visualizar o significado desses dígitos. Podemos usar o mesmo método esquemático para um número hexadecimal? Claro.

Considere o número D3h que mencionamos anteriormente. D é o dígito hexadecimal 13, e há 16 dígitos hexa, versus 10 para o decimal, de modo que D3h é 13 dezesseis e 3 unidades. Ou, de forma esquemática:

```
D  —> 13 x 16 = 208
 3 —>  3 x  1 =   3
-------------------
D3h           = 211
```

Para o número decimal 276, multiplicamos os dígitos por 100, 10 e 1; para o número hexa D3, multiplicamos os dígitos por 16 e 1. Se tivéssemos quatro dígitos decimais, teríamos multiplicado

por 1000, 100, 10 e 1. Quais os quatro números com que multiplicaríamos um número hexa de quatro dígitos?

Para decimal, os números 1000, 100, 10 e 1 são todos potências de 10:

10 elevado a 3 = 1000
10 elevado a 2 = 100
10 elevado a 1 = 10
10 elevado a 0 = 1

Podemos usar exatamente o mesmo método para os dígitos hexa, mas como potências de 16, ao invés de 10, de modo que os nossos quatro números seriam:

16 elevado a 3 = 4096
16 elevado a 2 = 256
16 elevado a 1 = 16
16 elevado a 0 = 1

Vamos converter o número 3AC8h em decimal usando os quatro números calculados acima:

3	—>	3 x 4096	=	12288
A	—>	10 x 256	=	2560
C	—>	12 x 16	=	192
8	—>	8 x 1	=	8
3AC8h			=	15048

7	—>	7 x 16	=	112
C	—>	12 x 1	=	12
7Ch			=	124

3	—>	3 x 256	=	768
F	—>	15 x 16	=	240
9	—>	9 x 1	=	9
3F9h			=	1.017

A	—>	10 x 4.096	=	40.960
F	—>	15 x 256	=	3.840
1	—>	1 x 16	=	16
C	—>	12 x 1	=	12
AF1Ch			=	44.828

3	—>	3 x 65.536	=	196.608
B	—>	11 x 4.096	=	45.056
8	—>	8 x 256	=	2.048
D	—>	13 x 16	=	208
2	—>	2 x 1	=	2
3B8D2h			=	243.922

Figura 1-3. ***Mais Conversões de Hexadecimal Para Decimal.***

Agora vamos descobrir o que acontece quando somamos números hexa que tenham mais do que um dígito. Para isto, usaremos o Debug e os números 3A7h e 1EDh:

```
-H 3A7 1ED
0594 01BA
```

Assim, vemos que 3A7h + 1EDh = 594h. Você pode checar os resultados convertendo estes números para decimal e fazendo a soma (e subtração, se quiser) em formato decimal; se você for mais aventuroso, faça os cálculos diretamente em hexa.

```
   1          1          1
  3A7         F451        C
+ 92A       + CB03      + D
  CD1         1BF54       19

  1111         1 1
   BCD8        BCD8
+  FAE9      + 0509
  1B7C1       C1E1
```

Figura 1-4. ***Mais Exemplos de Adição em Hexa***

Números Hexa de Cinco Dígitos

Até aqui, a matemática em hexa tem sido direta. O que acontece quando tentamos somar números hexa maiores? Vamos tentar um número hexa de cinco dígitos:

```
-H 5C3F0 4BC6
-        ^ Error
```

Esta é uma resposta inesperada. Por que o Debug diz que há um erro aqui? A razão tem a ver com uma unidade de armazenamento chamada *palavra*. A Hexaritmética do Debug trabalha somente com quatro dígitos hexa. Assim, se você tentar somar dois números hexa de quatro dígitos, tais como C000h e D000h (que deveria dar 19000h), você terá 9000h:

```
-H C000 D000
9000 F000
-
```

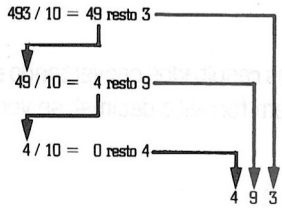
O Debug retém somente os quatro dígitos da direita da resposta.

Convertendo de Decimal Para Hexa

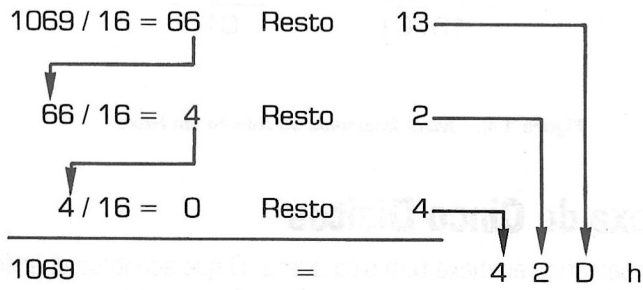
Até aqui vimos somente a conversão de hexa para decimal. Agora veremos como converter números decimais para hexa. Como já dissemos antes, no Capítulo 10 criaremos um programa para escrever os números do 8088 em decimal; no Capítulo 23 escreveremos outro programa para que o 8088 leia números decimais. Mas, assim como as conversões de hexa para decimal, vamos iniciar aprendendo como fazer as conversões a mão. Novamente, começaremos lembrando um pouco da matemática do colégio.

Quando aprendemos inicialmente a divisão, dividimos 9 por 2 e obtemos 4 com um resto de 1. Usaremos o resto para converter os números decimais para hexa.

Vejamos o que acontece quando dividimos repetidamente um número decimal, neste caso 493, por 10:



Os dígitos de 493 aparecem como restos em ordem contrária — isto é, começando com o da direita (3). Vimos na última seção que tudo o que precisávamos para a nossa conversão de hexa para decimal era trocar as potências de 10 por potências de 16. Para a conversão de decimal para hexa, podemos dividir por 16 ao invés de 10? Na verdade, este é o nosso método para a conversão.



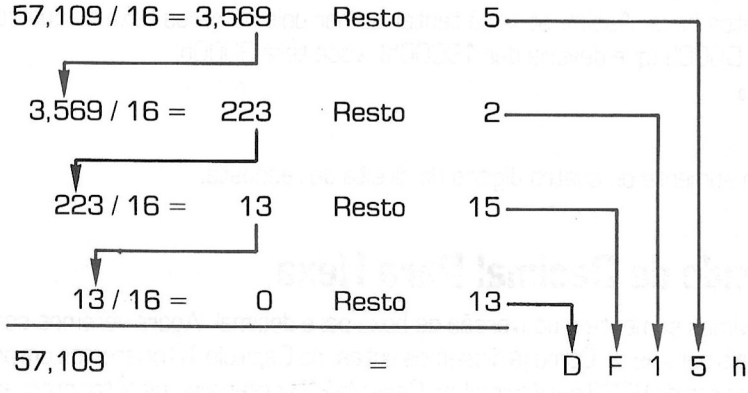
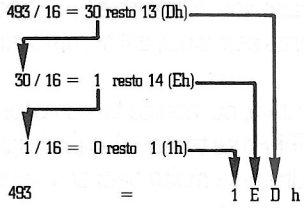


Figura 1-5. ***Mais Exemplos de Conversões Hexa.***

Por exemplo, vamos descobrir o número hexa equivalente a 493. Dividindo por 16, como vemos aqui:



Descobrimos que 1EDh é o equivalente hexa do decimal 493. Em outras palavras, vá dividindo por 16, e forme o número hexa final a partir dos restos. Isso é tudo.

Números Negativos

Se você ainda se lembra, temos um problema não resolvido com o número FFFFh. Dissemos que FFFFh é na verdade −1. E mais, se convertermos FFFFh para decimal, obteremos 65535. Como pode ser isto? Ele se comporta como um número negativo?

Bem, se adicionarmos FFFFh (ou seja, −1) a 5, o resultado seria 4, pois 5 − 1 = 4. E é isso o que acontece? Usando o comando H do Debug para somar 5 a FFFFh, nós encontramos:

```
-H 5 FFFF
0004 0006
-
```

O Debug *parece* tratar FFFFh como −1. FFFFh nem sempre se comportará como −1 nos programas que estivermos escrevendo. Para ver por que não, faça esta soma a mão.

Quando somamos dois números decimais, freqüentemente colocamos um "vai-um" na próxima coluna, assim:

```
  11
  95
+ 58
 153
```

A soma de dois números hexa não é muito diferente. Somar 3 a F nos dá 2, com um "vai-um" para a próxima coluna:

```
  1
  F
+ 3
 12h
```

Agora, veja o que acontece quando somamos 5 a FFFFh:

```
  1111
  0005h
+ FFFFh
 10004h
```

Como Fh + 1h = 10h, os "carries" (vai-uns) sucessivos se movem para as posições à esquerda. E se ignorarmos o último 1, temos a resposta correta para 5 − 1; a saber, 4. Por mais estranho que pareça, FFFFh se comporta como −1 quando ignoramos este estouro (*overflow*). Ele é chamado de overflow porque o número possui agora cinco dígitos, mas o Debug retém somente os primeiros quatro (os da direita).

Este overflow é um erro, ou a resposta está correta? Bem, sim e sim. Podemos escolher um dos dois. Mas as respostas não se contradizem? Não exatamente, pois podemos ver estes dois números de qualquer uma das formas.

Suponhamos que FFFFh seja igual a 65535. Ele é um número positivo, e é o maior número que podemos escrever com quatro dígitos hexa. Dizemos que FFFFh é um número *não sinalizado*. Ele não é sinalizado porque definimos os números de quatro dígitos como positivos. Somar 5 a FFFFh nos dá 10004h; nenhuma outra resposta seria correta. No caso de números sem sinal, então, um overflow é um erro.

Por outro lado, podemos tratar FFFFh como um número negativo, ou sinalizado, como foi feito pelo Debug quando usamos o comando H para somar FFFFh a 5. FFFFh se torna −1 quando ignoramos o *overflow*. De fato, os números de 8000h a FFFFh enquadram-se muito bem aos números negativos. Para números com sinal, como aqui, o overflow não é um erro.

O microprocessador 8088 pode admitir os números com ou sem sinal; a escolha é sua. Há instruções ligeiramente diferentes para os dois tipos, e veremos estas diferenças em outros capítulos quando começarmos a usar os números no 8088. Por enquanto, antes que você aprenda a escrever o negativo de, digamos, 3C8h, precisamos desmascarar o bit e ver como ele se encaixa ao esquema de bytes, palavras e hexa.

Bits, Bytes, Palavras e Notação Binária

Vamos agora nos aprofundar um pouco mais nos detalhes do seu IBM PC — é hora de aprendermos a aritmética do 8088: números binários. O microprocessador 8088, com toda a sua potência, é um tanto burro. Ele só conhece os dígitos 0 e 1, de modo que qualquer número usado por ele deve ser formado por uma grande fila de zeros e uns. Este é o sistema de numeração *binário* (ou sistema de base 2).

Quando o Debug imprime um número em hexa, ele usa um pequeno programa para converter seus números internos de binário para hexadecimal. No Capítulo 5, vamos montar um programa desse tipo para converter números binários para a notação hexadecimal, mas primeiro precisamos saber mais sobre os números binários.

Vamos pegar o número binário 1011b (o **b** significa binário). Este número é igual ao decimal 11, ou B em hexa. Para ver por quê, multiplique os dígitos de 1011b pelas potências da base 2:

Potências de 2:

$$2^3 = 8$$
$$2^2 = 4$$
$$2^1 = 2$$
$$2^0 = 1$$

Assim sendo,

$$1 \times 8 = 8$$
$$0 \times 4 = 0$$
$$1 \times 2 = 2$$
$$1 \times 1 = 1$$
$$1011b = 11 \text{ ou } Bh$$

Da mesma forma, 1111b é Fh, ou 15. E 1111b é o maior número sem sinal que pode ser colocado em quatro dígitos, enquanto 0000b é o menor. Assim, com quatro dígitos binários podemos escrever dezesseis números diferentes. Há exatamente 16 dígitos hexa, de modo que podemos escrever um dígitos hexa para cada quatro dígitos binários.

Um número hexa de dois dígitos, como 4Ch, pode ser escrito como 0100 1100b. Ele é composto de oito dígitos, que separamos em grupos de quatro por facilidade de leitura. Cada um destes dígitos binários é conhecido como *bit*, de modo que um número binário como 0100 1100b, ou 4Ch,

 possui oito bits.

Às vezes se torna importante numerar cada um dos bits em uma série longa, com o bit 0 sendo o mais à direita. O 1 no número 10b é o bit número 1, e o bit mais à esquerda de 1100b é o bit número 3. Ao numerarmos os bits desta forma, torna-se mais fácil qualquer referência a um bit individual, e iremos precisar disto mais adiante.

Um grupo de oito dígitos binários é conhecido como *byte*, e um grupo de 16 bits, ou dois bytes, é uma *palavra*. Usaremos estes termos diversas vezes no decorrer deste livro, pois os bits, bytes e palavras são peças fundamentais para o microprocessador 8088.

Binário	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Figura 1-6. ***Representação Binária, Hexa e Decimal de 0 a F.***

Agora, podemos ver por que a notação hexadecimal é tão conveniente; dois dígitos hexa encaixam-se perfeitamente em um byte (quatro bits para cada dígito hexa), e quatro dígitos ocupam exatamente uma palavra. Não podemos dizer o mesmo para os números decimais. Se tentássemos usar dois dígitos decimais para um byte, não poderíamos escrever números maiores do que 99, perdendo os valores de 100 a 255 — mais da metade da faixa dos números que um byte pode conter. E se usarmos três dígitos decimais, devemos ignorar mais da metade dos números decimais de três dígitos, pois os números de 256 até 999 não podem ser representados por um byte.

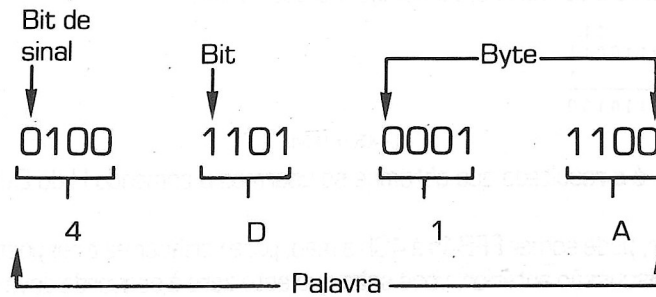


Figura 1-7. ***Uma Palavra é Formada por Bits e Bytes.***

Complemento de Dois — Um Tipo Estranho de Número Negativo

Agora estamos prontos para aprendermos mais sobre os números negativos. Dissemos que os números de 8000h a FFFFh comportam-se como números negativos quando ignoramos o "overflow". Há um modo fácil de reconhecer os números negativos quando escrevemos em binário:

Números Positivos:	
0000h	0000 0000 0000 0000b
•	•
•	•
•	•
7FFFh	0111 11111111 1111b
Números Negativos:	
8000h	1000 0000 0000 0000b
•	•
•	•
•	•
FFFFh	1111 11111111 1111b

Em todas as formas binárias dos números positivos, o primeiro bit (bit 15) é sempre 0. Para todos os números negativos, o primeiro bit é sempre 1. Esta diferença, de fato, é o modo com que o 8088 reconhece um número negativo: ele olha para o bit 15, o *bit de sinal*. Se usarmos instruções para números sem sinal em nossos programas, o 8088 ignorará o bit de sinal, e podemos usar os números sem sinal conforme nossa conveniência.

Estes números negativos são conhecidos como *Complemento de Dois* dos números positivos. Por que complemento? Porque a conversão de um número positivo, como 3C8h, para sua forma de complemento de dois é um processo em dois passos, sendo o primeiro a conversão do número para o seu *complemento*.

Não precisaremos muito dos números negativos, mas faremos a conversão aqui só para você ver como o 8088 "nega" os números. A conversão vai parecer um pouco estranha. Você não verá por que ela funciona, mas verá que funciona!

Para encontrar a forma de complemento de dois (negativo) de qualquer número, primeiro escreva o número em binário, ignorando o sinal. Por exemplo, 4Ch torna-se 0000 0000 0100 1100b.

Para tornar esse número negativo, primeiro inverta todos os zeros e uns. Este processo de inversão é chamado *complementação*, e tomando o complemento de 4Ch, encontramos o seguinte:

```
0000000001001100
```

torna-se:

```
1111 1111 1011 0011
```

No segundo passo para a conversão, somamos 1 a esse valor:

```
                  11
  1111111110110011
+                  1
------------------
  1111111110110100
```

(−4Ch = FFB4h)

A resposta, FFB4h, é o resultado que obtemos se usarmos o comando H do Debug para subtrair 4Ch de 0h.

Se você quiser, pode somar FFB4h a 4Ch a mão, para verificar se a resposta é 10000h. E de acordo com nossa discussão anterior, você sabe que este zero à esquerda deve ser ignorado para obter 0 (4C + (−4C) = 0) quando você fizer a soma em complemento de dois.

Resumo

Este capítulo foi um grande passo em direção ao mundo hexadecimal e aos números binários, e certamente foi utilizado muito esforço mental. Logo, no Capítulo 3, iremos diminuir um pouco o passo — uma vez que você já aprendeu o bastante para lidar com o Debug. Agora, vamos tomar um fôlego e ver o que já aprendemos e o que descobrimos.

Começamos descobrindo o Debug. Nos capítulos a seguir, nos tornaremos amigos íntimos do Debug mas, como ele não entende o nosso sistema decimal tão comum, começaremos nossa amizade aprendendo um pouco mais sobre um novo sistema de numeração, a notação hexadecimal.

Aprendendo sobre números hexa, você também aprendeu a converter os números de decimal para hexa, e vice-versa. Mais adiante escreveremos um programa para fazer estas conversões, mas por enquanto é necessário aprender a própria linguagem.

Tendo visto as bases da notação hexadecimal, já pudemos descobrir os bits, bytes, palavras e números binários — personagens importantes que você encontrará freqüentemente quando estivermos estudando o mundo do 8088 e da programação em linguagem assembly.

Finalmente, continuamos vendo os números negativos em hexa — os números em complemento de dois. Eles nos levaram aos números com e sem sinal, onde também vimos overflows de dois tipos diferentes: um em que o overflow leva a uma resposta correta (adição de dois números com sinal), e um em que o overflow leva a uma resposta errada (adição de dois números sem sinal).

Todo este aprendizado será útil nos capítulos seguintes, pois usaremos nosso conhecimento de números hexa para falar com o Debug, e o Debug atuará como um interpretador entre nós e o microprocessador 8088 dentro do seu IBM PC.

No capítulo seguinte, usaremos o conhêcimento que obtivemos até aqui para aprender mais sobre o 8088. Novamente confiaremos no Debug, e usaremos números hexa, ao invés de binários, para falar com o 8088. Aprenderemos sobre os registradores do 8088 — os lugares onde ele armazena os números — e, no Capítulo 3, estaremos prontos para escrever um programa real que imprimirá um caractere na tela. Também aprenderemos mais sobre o modo com que o 8088 realiza sua matemática; quando atingirmos o Capítulo 10, já poderemos escrever um programa para converter números binários em decimais.

2

ARITMÉTICA 8088

Registradores Como Variáveis

A Memória e o 8088

Adição ao Modo 8088

Subtração ao Modo 8088

Números Negativos no 8088

Bytes no 8088

Multiplicação e Divisão ao Modo 8088

Resumo

Sabendo algo sobre a aritmética hexa do Debug e da aritmética binária do 8088, podemos começar a aprender como o 8088 realiza sua matemática. Ele usa comandos internos chamados *instruções*.

Registradores Como Variáveis

O Debug, nosso guia e intérprete, sabe muito sobre o 8088 que está dentro do seu IBM PC. Iremos usá-lo para entrarmos no mundo interior do 8088, e começaremos a pedir ao Debug para apresentar o que puder sobre pequenas partes de memória chamadas *registradores*, em que podemos armazenar dados. Os registradores são como variáveis em BASIC, mas não são exatamente os mesmos. Diferente da linguagem BASIC, o microprocessador 8088 contém um número fixo de registradores, e eles não fazem parte da memória do seu IBM PC.

Pediremos ao Debug para apresentar os registradores do 8088 com o comando R, de *Registrador*:

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 E485            IN          AL,85
```

(Você provavelmente verá números diferentes na segunda e terceira linhas da sua listagem; esses números refletem a quantidade de memória em um computador. Você continuará vendo tais diferenças, e depois aprenderemos mais sobre elas.)

Por enquanto, o Debug certamente nos deu muita informação. Vamos nos concentrar nos primeiros quatro registradores, AX, BX, CX e DX, todos os quais o Debug nos diz serem iguais a 0000, tanto aqui quanto em sua tela. Estes registradores são os *registradores de uso geral*. Os outros registradores, SP, BP, SI, DI, DS, ES, SS, CS e IP, são registradores de propósito especial que veremos em outros capítulos.

O número de quatro dígitos que vem após o nome do registrador está em notação hexa. No Capítulo 1, aprendemos que uma palavra é descrita exatamente em quatro dígitos hexa. Aqui, você pode ver que cada um dos 13 registradores do 8088 possui uma palavra, ou 16 bits de tamanho. Isto é porque os computadores baseados no microprocessador 8088 são conhecidos como máquinas de 16 bits.

Já dissemos que os registradores são como variáveis BASIC. Isto deve significar que podemos alterá-las, e realmente podemos. O comando R do Debug faz mais do que apresentar registradores. Seguido pelo nome de um registrador, o comando diz ao Debug que queremos ver o registrador, e depois alterá-lo. Por exemplo, podemos mudar o registrador AX assim:

```
-R AX
AX 0000
:3A7
```

Vejamos os registradores novamente para ver se o registrador AX contém agora 3A7h:

```
-R
AX=03A7 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 E485            IN          AL,85
```

Tudo certo. Além do mais, podemos colocar qualquer número hexa em qualquer registrador com o comando R especificando somente o nome do registrador e entrando o novo número após os dois-pontos, e pronto. Daqui em diante, usaremos este comando sempre que precisarmos colocar números nos registradores do 8088.

Você deve se lembrar de ter visto o número 3A7h no Capítulo 1, onde usamos o comando Hexaritmético do Debug para somar 3A7h e 1EDh. Lá, o Debug fez o trabalho para nós. Desta vez, usaremos o Debug simplesmente como um interpretador para que possamos trabalhar diretamente com o 8088. Daremos ao 8088 as instruções para somar dois registradores: colocaremos um número no registrador BX e depois instruiremos ao 8088 para somar o número em BX ao número em AX e colocar a resposta de volta em AX. Primeiro, precisamos de um número no registrador BX. Desta vez, vamos somar 3A7h e 92Ah. Use o comando R para armazenar 92Ah em BX.

A Memória e o 8088

Os registradores AX e BX devem, respectivamente, conter 3A7h e 92Ah, como podemos verificar com o comando R:

```
-R
AX=03A7 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 E485          IN          AL,85
```

Agora que temos nossos dois números nos registradores AX e BX, como podemos dizer ao 8088 para somar BX a AX? Colocamos alguns números na memória do computador.

Seu IBM PC provavelmente possui pelo menos 128K de memória — muito mais do que o necessário aqui. Colocaremos dois bytes de *código de máquina* em algum lugar nesta imensidão de memória. Neste caso, o código de máquina serão dois números binários que dirão ao 8088 para somar o registrador BX a AX. Depois, para que possamos ver o que acontece, *executaremos* esta instrução com a ajuda do Debug.

Agora, em que lugar da memória iremos colocar nossa instrução de dois bytes, e como dizer ao 8088 onde encontrá-la? Por acaso, o 8088 divide sua memória em partes de 64K chamadas de *segmentos*. Na maior parte do tempo, veremos a memória dentro de um desses segmentos sem que realmente saibamos onde começa esse segmento. Podemos fazer isto pelo modo com que o 8088 rotula a memória.

Todos os bytes na memória são rotulados com números começando de 0 e subindo. Mas você se lembra da limitação de quatro dígitos para os números hexa? Isto significa que o maior número que o 8088 pode usar é o equivalente hexa a 65535, o que significa que o número máximo de rótulos que ele pode usar é 64K. Mesmo assim, sabemos por experiência que o 8088 pode fazer referência a mais de 64K de memória. Como pode ser isso? Com um pequeno truque: ele usa dois números, um para cada segmento de 64K, e um para cada byte, ou *offset* (deslocamento) dentro de cada segmento. Cada segmento começa em um múltiplo de 16 bytes, de forma que sobrepondo segmentos de offsets, o 8088 pode rotular efetivamente bem mais do que 64K de memória. De fato, é exatamente por isso que o 8088 usa até um milhão e poucos bytes de memória.

Todos os endereços (rótulos) que usaremos são offsets, ou deslocamentos desde o início do segmento. Por exemplo, 3756:0100 significará que estamos em um offset de 100h dentro do segmento 3756h.

Depois, no Capítulo 11, veremos mais sobre segmentos e por que temos um número de segmento tão alto. Mas, por enquanto, confiaremos no Debug para escolher os segmentos para nós, para que possamos trabalhar dentro de um segmento sem termos que prestar atenção no seu número. E, por enquanto, iremos nos referir aos endereços somente pelos seus offsets. Cada um destes endereços refere-se a um byte no segmento, e os endereços são seqüenciais, de forma que 101h é o byte seguinte a 100h na memória.

Escritas por inteiro, nossas duas instruções para somar BX a AX se parece assim: ADD AD,BX. Colocaremos estas instruções nas posições 100h e 101h, em qualquer segmento que o Debug escolha para usar. O Debug começa a operar. Em se tratando de nossa instrução de soma (ADD), diremos que ela está na posição 100h, pois esta é a localização do primeiro byte da instrução.

O comando do Debug para examinar e alterar a memória é chamado de E, de *Enter*. Use este comando para entrar os dois bytes da instrução ADD, como se segue:

```
-E 100
3756:0100 E4.01
 E 101
3756:0101 85.D8
```

Os números 01h e D8h são os comandos em linguagem de máquina para nossa instrução ADD nas posições de memória 3756:0100 e 3756:0101. O número de segmento que você verá provavelmente será outro, mas essa diferença não afetará nosso programa. Da mesma forma, o Debug provavelmente apresentará um outro número de dois dígitos para cada um dos comandos E. Estes números (E4h e 85h em nosso exemplo) são os números antigos na memória nos offsets 100h e 101h do segmento que o Debug escolheu — isto é, os números são dados de programas anteriores deixados na memória quando você iniciou com o Debug. (Se você acabou de ligar o computador, os números devem ser 00.)

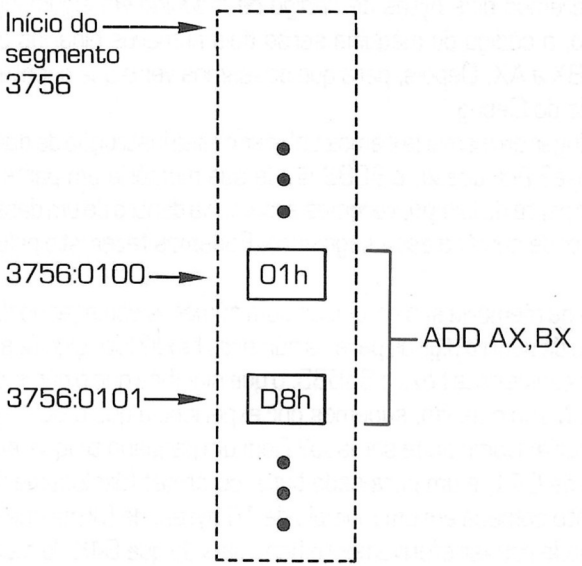


Figura 2-1. ***Nossa Instrução se Inicia 100h Bytes do Início do Segmento***

Adição ao Modo 8088

Agora a sua listagem deve ser semelhante a esta:

```
-R
AX=03A7 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 01D8              ADD             AX,BX
```

Nossa instrução ADD está corretamente colocada na memória, justamente onde queremos que ela esteja. Os primeiros dois números da última linha, 3756:0100, nos dão o endereço (100h) do primeiro número da nossa instrução ADD. A seguir, vemos os dois bytes de ADD: 01D8. O byte igual a 01h está no endereço 100h, enquanto D8h está em 101h. Finalmente, como entramos nossa instrução em *linguagem de máquina* — números que não têm significado para nós, mas que o 8088 interpretará como uma instrução de soma —, a mensagem ADD AX,BX confirma que entramos a instrução corretamente.

Embora tivéssemos colocado nossa instrução ADD na memória, não estamos prontos para rodá-la (*executá-la*) por meio do 8088. Primeiro, precisamos dizer ao 8088 onde encontrar a instrução.

O 8088 encontra o segmento e o deslocamento em dois registradores especiais, CS e IP, que você pode ver na listagem acima. O número do segmento é guardado em CS, ou *Code Segment* (*segmento de código*), um registrador especial que iremos ver melhor mais adiante. Se você olhar na listagem, verá que o Debug já definiu o segmento para nós (CS=3756, em nosso exemplo). O endereço completo de nossa instrução, no entanto, é 3756:0100.

A segunda parte deste endereço (o offset dentro do segmento 3756) é guardado no registrador IP, de *Instruction Pointer* (*ponteiro de instrução*). O 8088 usa o offset no registrador IP para descobrir nossa primeira instrução. Podemos dizer a ele onde encontrá-la definindo um valor para o registrador IP — IP=0100.

Mas o registrador IP já está como 100h. Fomos espertos: o Debug sempre coloca IP em 100h quando inicia seu trabalho. Sabendo disto, escolhemos deliberadamente 100h como o endereço da nossa primeira instrução e eliminamos a necessidade de atribuir um valor a IP em um passo separado. Esse é um bom ponto para se ter sempre em mente.

Agora, com nossa instrução no lugar e os registradores com os valores corretos, diremos ao Debug para executar nossa única instrução. Usaremos o comando T do Debug (de *Trace*), que executa uma instrução de cada vez e depois apresenta os registradores. Após cada passo, o IP deve apontar para a próxima instrução. Neste caso, ele apontará para 102h. Não colocamos uma instrução em 102h, de modo que veremos na última linha da apresentação dos registradores uma instrução deixada por algum outro programa.

Vamos pedir ao Debug para executar uma instrução com o comando T:

```
-T
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0102   NV UP DI PL NZ AC PE NC
3756:0102 AC                LODSB
```

E é isso aí. O registrador AX agora contém CD1h, que é a soma de 3A7h e 92Ah. E o registrador IP aponta para o endereço 102h, de modo que a última linha acima mostra alguma instrução na posição de memória 102h, ao invés da 100h.

Já dissemos antes que o ponteiro de instrução, juntamente com o registrador CS, sempre aponta para a próxima instrução para o 8088. Se digitássemos T novamente, executaríamos a instrução seguinte, mas não faça isso ainda — seu 8088 pode perder a cabeça.

Ao invés disso, que tal se executássemos a instrução de soma novamente, somando 92Ah a CD1h e armazenando a nova resposta em AX? Para isso, precisamos dizer ao 8088 onde encontrar sua nova instrução, e queremos que ela seja nossa instrução ADD em 0100h. Podemos simplesmente trocar o registrador IP para 0100h? Vamos tentar. Use o comando R para colocar 100 em IP, e olhe como ficaram os registradores:

```
AX=0CD1 BX=092A CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3756 ES=3756 SS=3756 CS=3756 IP=0100   NV UP DI PL NZ AC PE NC
3756:0100 ADD               AX,BX
```

Conseguiu. Tente o comando T novamente e veja se o registrador AX contém 15FBh. Perfeito!

Como você pode ver, você deve sempre checar o registrador IP e a instrução na linha final da listagem de R antes de usar o comando T. Dessa forma, você estará seguro de que o 8088 irá executar a instrução que você quer.

Agora, coloque 100h novamente no registrador IP, e assegure-se de que os registradores contêm AX = 15FB, BX = 092A, e vamos tentar a subtração.

Figura 2-2. ***Antes de Executar a Instrução ADD.***

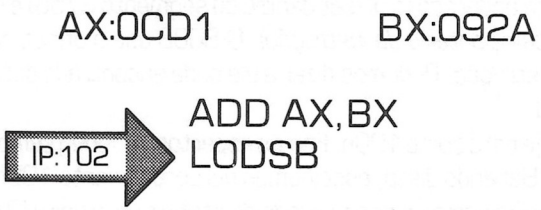


Figura 2-3. ***Após a Execução da Instrução ADD.***

Subtração ao Modo 8088

Vamos escrever uma instrução para subtrair BX de AX de modo que, após a subtração, teremos 3A7h em AX: o ponto em que começamos antes das duas somas. Você também verá como economizar um pouco de trabalho para entrar com mais de um byte na memória.

Quando entramos os dois bytes para a instrução ADD, digitamos o comando E duas vezes: uma com 0100h para o primeiro endereço, e outra com 0101h para o segundo endereço. Esta rotina funcionou, mas, como veremos, podemos entrar com o segundo byte sem outro comando se o separarmos do primeiro byte por meio de um espaço. Quando você tiver terminado de entrar os bytes, pressione a tecla Enter, e você sairá do comando de entrada. Tente este método em nossa instrução de subtração:

```
-E 100
3756:0100 01.29 D8.D8
```

A listagem dos registradores (lembre-se de passar IP para 100h) deve agora mostrar a instrução *SUB AX,BX*, que instrui o 8088 a subtrair o conteúdo de BX do conteúdo de AX, e deixar o resultado em AX. A ordem de AX e BX pode parecer invertida, mas a instrução é como o comando BASIC AX = AX – BX, exceto que no 8088, diferente do BASIC, sempre o resultado é colocado na primeira variável (registrador).

Execute esta instrução com o comando T do Debug. AX deve conter CD1. Mude IP para apontar novamente para esta instrução, e execute o comando novamente (lembre-se de checar a instrução logo abaixo dos registradores). AX deve agora ser 03A7.

Números Negativos no 8088

No último capítulo, aprendemos como o 8088 usa a forma de complemento de dois para números negativos. Agora, vamos trabalhar diretamente com a instrução SUB para calcular os números negativos. Vamos testar um pouco o 8088, para ver se obtemos FFFF para −1. Vamos subtrair um de zero e, se estivermos certos, a subtração deve colocar FFFF (−1) em AX. Coloque 0 em AX e 1 em BX, e depois execute a instrução no endereço 100h. Bem que eu falei: AX = FFFFh.

Enquanto você estiver com esta instrução à mão, você poderá tentar alguns números diferentes para ficar mais acostumado à aritmética complemento de dois. Por exemplo, veja que resultado você obtém para −2.

Bytes no 8088

Toda a nossa aritmética até aqui foi executada em palavras, daí os quatro dígitos hexa. O microprocessador 8088 pode executar aritmética com bytes? Sim, senhor.

Como uma palavra é formada por dois bytes, cada registrador de uso geral pode ser dividido em dois bytes, conhecidos como *byte alto* ou *"High"* (os primeiros dois dígitos hexa) e o *byte baixo* ou *"Low"* (os dois dígitos hexa seguintes). Cada um destes registradores pode ser chamado pela sua letra (de A até D), seguida por X para uma palavra, H para o byte alto, ou L para o byte baixo. Por exemplo, DL e DH são registradores de byte, enquanto DX é um registrador de palavra. (Esta terminologia pode-se tornar confusa mais tarde, pois as palavras armazenadas na memória possuem o byte baixo em primeiro lugar, e depois o byte alto.)

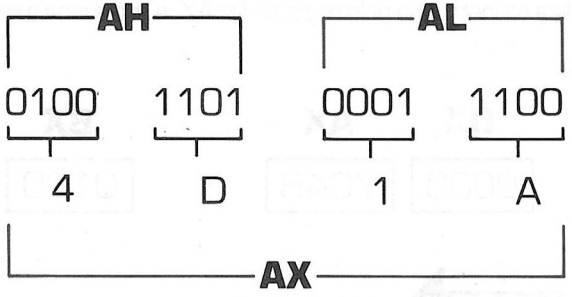


Figura 2-4. ***O Registrador AX Dividido em Dois Registradores de Byte (AH e AL).***

Vamos testar a matemática de byte com uma instrução ADD. Entre os dois bytes 00h e C4h, iniciando na posição 0100h. Abaixo do conteúdo dos registradores, você verá a instrução *ADD AH,AL*, que somará os dois bytes do registrador AX e colocará o resultado no byte alto, AH.

A seguir, carregue o registrador AX com 0102h. Isto colocará 01h em AH e 02h em AL. Defina o registrador IP como 100h, execute o comando T, e então descobrirá que AX agora contém 0302. O resultado de 01h + 02h é 03h, e este valor está agora no registrador AH.

Mas suponha que você não quisesse somar 01h e 02h. Suponha que você realmente quisesse somar 01h e 03h. Se o registrador AX já estivesse com 0102, você poderia mudar o registrador AL para 03? Não. Você teria que mudar o registrador AX para 0103h. Por quê? Porque o Debug somente nos permite mudar registradores de palavra *inteira*. Não há uma forma de se mudar somente a parte alta ou baixa de um registrador com o Debug. Mas, como você viu no capítulo anterior, isto

não é problema. Com os números hexa, podemos dividir uma palavra em dois bytes partindo o número hexa de quatro dígitos ao meio. Assim, a palavra 0103h torna-se dois bytes 01h e 03h.

Para tentar esta instrução ADD, carregue o registrador AX com 0103h. Sua instrução ADD AH,AL ainda está na memória na posição 0100h; então, passe o registrador IP para 100h e, com 01h e 03h agora presentes nos registradores AH e AL, execute esta instrução. Desta vez, AX conterá 0403h: 04h, a soma de 01h e 03h, está agora no registrador AH.

Multiplicação e Divisão ao Modo 8088

Vimos o 8088 somando e subtraindo dois números. Agora veremos que ele também pode multiplicar e dividir — inteligente, ele. A instrução para multiplicar é chamada MUL, e o código de máquina para multiplicar AX e BX é F7h E3h. Entraremos com isto na memória, mas primeiro vamos dar uma palavrinha sobre a instrução MUL.

Onde a instrução MUL armazena seu resultado? No registrador AX? Não exatamente; devemos ter cuidado aqui. Como você logo verá, a multiplicação de dois números de 16 bits pode dar uma resposta de 32 bits, de modo que a instrução MUL guarda o seu resultado em dois registradores, DX e AX. Os 16 bits mais altos (ou mais significativos) são colocados no registrador DX; os mais baixos, em AX. Podemos também escrever esta combinação de registradores como DX:AX.

Vamos voltar ao Debug e ao 8088. Entre com a instrução de multiplicação, F7h E3h, na posição 0100h, assim como você fez para as instruções de soma e subtração, e defina AX = 7C4Bh e BX = 100h. Você verá que a instrução na linha de baixo irá ser *MUL BX*, sem qualquer referência ao registrador AX. Para multiplicar palavras, como estamos fazendo, o 8088 *sempre* multiplica o registrador que você indica na instrução pelo registrador AX, e armazena o resultado no par de registradores DX:AX.

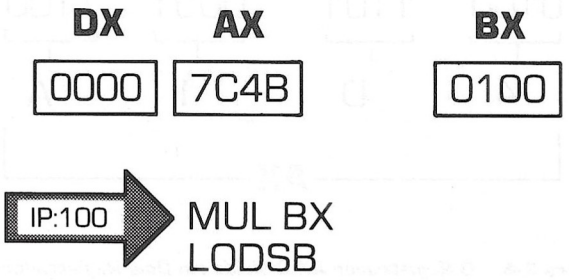


Figura 2-5. ***Antes de Executar a Instrução MUL.***

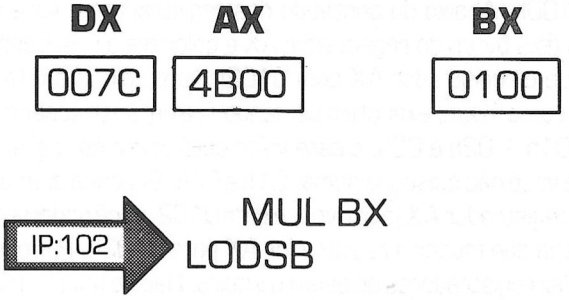


Figura 2-6. ***Depois de Executar a Instrução MUL.***

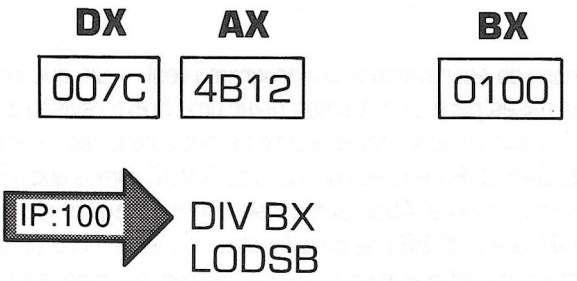


Figura 2-7. ***Antes de Executar a Instrução DIV.***

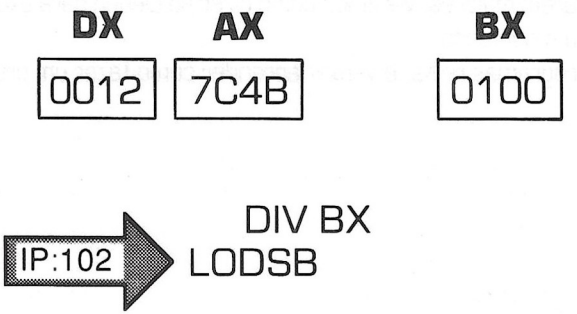


Figura 2-8. ***Depois de Executar a Instrução DIV.***

Antes de executarmos a instrução MUL, vamos fazer a multiplicação a mão. Como vamos calcular 100h * 7C4B? Os três dígitos 100 possuem o mesmo efeito em decimal e hexadecimal. Portanto, a multiplicação por 100h significa somente acrescentar dois zeros à direita de um número hexa. Assim, 100h * 7C4B = 7C4B00h. Este resultado é muito grande para ficar em só uma palavra, e iremos dividi-lo nas duas palavras 007Ch e 4B00h.

Use o Debug para executar a instrução. Você verá que DX contém a palavra 007Ch, e AX contém a palavra 4B00h. Em outras palavras, o 8088 retornou o resultado da instrução de *multiplicar palavra* no par de registradores DX:AX. Como a multiplicação de duas palavras nunca será maior do que duas palavras, mas geralmente será maior do que uma palavra (como acabamos de ver), a instrução de multiplicar palavras sempre retorna o resultado no par de registradores DX:AX.

E a divisão? Quando dividimos os números, o 8088 mantém o quociente e o resto da divisão. Vejamos como funciona a divisão do 8088. Primeiro coloque a instrução F7h F3h em 0100h (e 0101h). Assim como a instrução MUL, DIV usa DX:AX sem que isso seja dito, de modo que tudo que veremos será DIV BX. Agora, carregue os registradores de modo que DX = 007C e AX = 4B12h; BX deve conter, ainda 0100h.

Novamente, calcularemos o resultado a mão: 7C4B12h / 100h = 7C4Bh, com um resto de 12h. Quando executarmos nossa instrução de divisão em 0100h, veremos que AX = 7C4Bh, o resultado de nossa divisão, e DX = 0012h, que é o resto. (Veremos um ótimo uso para este resto no Capítulo 10, quando escreveremos um programa para converter números decimais para hexa, usando o resto, assim como fizemos no Capítulo 1.)

Resumo

Está quase na hora de escrevermos um programa real — um que imprima um caractere na tela. Até aqui vimos as bases para isso. Vamos fazer uma breve revisão e seguiremos e frente.

Começamos este capítulo aprendendo sobre os registradores e verificando suas semelhanças com as variáveis do BASIC. Porém, ao contrário do BASIC, vimos que o 8088 possui um número pequeno e limitado de registradores. Abordamos mais sobre os registradores de uso geral, com uma breve olhada em CS e IP, que o 8088 usa para localizar os endereços de segmento e offset.

Após termos visto como alterar e ler os registradores, passamos para pequenos programas de uma única instrução entrando com os códigos de máquina para somar, subtrair, multiplicar e dividir dois números por meio dos registradores AX e BX. Mais adiante iremos precisar muito do que aprendemos aqui, mas você não precisará decorar os códigos de máquina das instruções.

Também vimos como dizer ao Debug para executar uma única instrução. Mais adiante, quando os programas se tornarem maiores, veremos como dizer ao Debug para executar mais de uma instrução com somente um comando.

Voltemos aos programas reais, e vamos aprender como fazer um programa que fala.

3

IMPRIMINDO CARACTERES

INT — A Poderosa Interrupção

Uma Boa Saída — INT 20h

Um Programa de Duas Linhas — Juntando as Partes

Entrando Programas

MOVendo Dados Para os Registradores

Imprimindo um String de Caracteres

Resumo

Agora, já sabemos o bastante para fazer algo mais sólido; portanto, levante as mangas e estique os dedos. Vamos começar instruindo o DOS a enviar um caractere para a tela, e depois passaremos para um trabalho mais interessante. Montaremos um pequeno programa com mais de uma instrução, e daqui por diante, aprender outro modo de colocar dados nos registradores — desta vez, de dentro de um programa. Agora, vejamos se podemos fazer o DOS falar.

INT — A Poderosa Interrupção

Às nossas quatro instruções de matemática, ADD, SUB, MUL e DIV, juntaremos uma nova instrução chamada INT (de *Interrupção*). INT é algo como o GOSUB do BASIC. Usaremos a instrução INT para pedir ao DOS para imprimir um caractere, A, na tela para nós.

Antes de aprender como INT funciona, vamos ver um exemplo. Inicie o Debug e coloque 200h em AX e 41h em DX. A instrução INT para as funções do DOS é INT 21h — em código de máquina, CDh 21h. Esta é uma instrução de dois bytes como a instrução DIV no capítulo anterior. Coloque INT 21h na memória, começando na posição 100h, e use o comando R para confirmar que esta instrução é INT 21 (lembre-se de definir IP como 100h se já não estiver lá).

Agora estamos prontos para executar esta instrução, mas não podemos usar o comando T aqui como fizemos da última vez. O comando T executa uma instrução de cada vez, mas a instrução INT chama um grande programa do DOS para fazer o trabalho, assim como os programas em BASIC podem chamar uma sub-rotina com o comando GOSUB.

Não queremos acompanhar a execução de cada uma das instruções da "sub-rotina" inteira do DOS. Ao invés disso, queremos executar nosso programa de uma linha, mas parar antes de executar a instrução na posição 102h. Podemos fazer isto com o comando G do Debug ("G" de Go — seguir), seguido pelo endereço em que queremos parar:

```
-G 102
A
AX=0241 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0102   NV UP DI PL NZ NA PO NC
3970:0102 8BE5              MOV     SP,BP
```

O DOS imprimiu um caractere A, e depois passou o controle para nosso pequeno programa. (Lembre-se: a instrução em 102h é somente algum dado deixado por outro programa, e você provavelmente verá algo diferente.)

Nosso pequeno programa aqui possui, de certa forma, duas instruções, com a segunda instrução localizada em 102h, isto é, ele é algo assim:

```
INT     21
MOV     SP,BP       (Ou qualquer outra coisa que esteja presente)
```

Logo trocaremos esta segunda instrução qualquer por uma nossa. Por enquanto, como ela não é algo que queremos executar, dissemos ao Debug para executar nosso programa, parar a execução quando atingir a segunda instrução, e apresentar os registradores quando tiver terminado.

E como o DOS soube que devia imprimir A? O 02h no registrador AH disse ao DOS para imprimir um caractere. Outro número em AH diria ao DOS para executar uma função diferente. (Veremos outras depois, mas se você quer saber agora, você pode encontrar uma lista das funções em seu *Manual Técnico do DOS*.)

Quanto ao caractere em si, o DOS usa o número no registrador DL como o código ASCII do caractere a imprimir quando o pedirmos para enviar um caractere à tela. Nós colocamos 41h, o código ASCII para a letra A maiúscula.

No Apêndice E, você encontrará uma tabela dos códigos ASCII dos caracteres que o IBM PC pode apresentar. Por nossa conveniência, os números estão em decimal e hexadecimal. Mas como o

Debug lê somente em hexa, eis aqui uma boa chance para você praticar a conversão de números decimais para hexa. Apanhe um caractere da tabela e o converta para hexa sozinho. Depois verifique sua conversão digitando seu valor hexa no registrador DL e executando a instrução INT novamente (lembre-se de resetar IP para 100h).

Você deve querer saber o que aconteceria se tivesse tentado o comando T na instrução INT. Vamos imaginar que não executamos ainda o comando G 102 e, no lugar disso, vamos acompanhar o programa por um certo tempo, para ver o que acontece. Se você quiser tentar sozinho, não vá muito longe: seu IBM PC pode fazer algo estranho. Após ter acompanhado alguns passos, saia do Debug com o comando Q. Isto limpará qualquer mensagem que você deixou para trás.

```
-R
AX=0200 BX=0000 CX=0000 DX=0041 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100   NV UP DI PL NZ NA PO NC
3970:0100 CD21                INT       21
-T

AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0180   NV UP DI PL NZ NA PO NC
3372:0180 80FC4B              CMP       AH,4B
-T

AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0183   NV UP DI NG NZ AC PE CY
3372:0183 7405                JZ        018A
-T

AX=0200 BX=0000 CX=0000 DX=0041 SP=FFE8 BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3372 IP=0185   NV UP DI NG NZ AC PE CY
3372:0185 2E                  CS:
3372:0185 FF2EAB0B            JMP       FAR [0BAB]          CS=0BAB:0BFF
-Q
```

Note que o primeiro número do endereço mudou aqui, de 3970 para 3372. Estas últimas três instruções foram parte do DOS, e o programa para o DOS está em outro segmento. Na verdade, há muitas outras instruções que o DOS executa antes de imprimir uma única letra A na tela; mesmo uma tarefa aparentemente tão simples não é tão fácil como parece. Agora você pode ver por que eu usei o comando G para executar nosso programa até a posição 102h. De outra forma, veríamos uma grande quantidade de instruções do DOS. (Se você estiver usando uma versão do DOS diferente da que eu usei, as instruções que você verá podem ser um pouco diferentes.)

Uma Boa Saída — INT 20h

Lembra-se de que nossa instrução INT foi 21h? Se mudássemos o 21h para um 20h, teríamos a instrução INT 20h. Essa é uma outra instrução de interrupção, e diz ao DOS que queremos sair do nosso programa, para que o DOS possa tomar controle total novamente. Em nosso caso, INT 20h passará o controle de volta ao Debug, pois estamos executando nossos programas pelo Debug, e não pelo DOS.

Entre com a instrução CDh 20h, começando na posição 100h, e depois tente o seguinte (lembre-se de checar a instrução INT 20 com o comando R):

```
-G 102

Program terminated normally
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100   NV UP DI PL NZ NA PO NC
3970:0100 CD20                INT       20
-G
```

```
Program terminated normally
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=3970 ES=3970 SS=3970 CS=3970 IP=0100   NV UP DI PL NZ NA PO NC
3970:0100 CD20              INT         20
```

O comando G, com nenhum número a seguir, executa o programa inteiro (que possui somente uma instrução agora, pois INT 20 é uma instrução de *saída*), e então retorna ao início. IP foi resetado para 100h, onde havíamos começado. Os registradores neste exemplo são zero só porque começamos com o Debug agora.

Podemos usar a instrução INT 20 ao final de um programa para retornar o controle de um modo gracioso ao DOS (ou Debug), de forma que iremos juntar esta instrução com INT 21h para formar um programa de duas linhas.

Um Programa de Duas Linhas — Juntando as Partes

Começando na posição 100h, entre as duas instruções INT 21h, INT 20h (CD 21 CD 20) uma após a outra. (Daqui por diante, sempre começaremos os programas na posição 100h).

Quando tínhamos somente uma instrução podíamos "listar" essa instrução com o comando R, mas agora temos duas instruções. Para vê-las, temos o comando U (*Unassemble — desassemblar*), que atua como o comando LIST do BASIC:

```
-U 100
3970:0100 CD21          INT     21
3970:0102 CD20          INT     20
3970:0104 D98D460250B8  ESC     09,[DI+0246][DI+B850]
3970:010A 8D00          LEA     AX,[BX+SI]
3970:010C 50            PUSH    AX
3970:010D E82A23        CALL    243A
3970:0110 8BE5          MOV     SP,BP
3970:0112 83C41A        ADD     SP,+1A
3970:0115 5D            POP     BP
3970:0116 C3            RET
3970:0117 55            PUSH    BP
3970:0118 83EC02        SUB     SP,+02
3970:011B 8BEC          MOV     BP,SP
3970:011D 823E0E0000    CMP     BYTE PTR [000E],00
```

As primeiras duas instruções reconhecemos como as duas instruções que acabamos de entrar. As outras instruções são reminiscências deixadas na memória. Conforme nosso programa cresce, a tela irá conter mais código nosso.

Agora preencha o registrador AH com 02h e o registrador DL com o número para qualquer caractere (assim como você fez antes quando alterou os registradores AX e DX), e depois simplesmente digite o comando G para ver o seu caractere. Por exemplo, se você colocar 41h em DL, você verá:

```
-G
A
Program terminated normally
-
```

Tente isto algumas vezes antes de continuarmos vendo outros modos de setar estes registradores.

Entrando Programas

Daqui por diante, a maioria dos nossos programas terá mais do que uma instrução, e para apresentar estes programas iremos usar uma listagem desassemblada. Nosso último programa apareceria desta forma:

```
3970:0100 CD21          INT     21
3970:0102 CD20          INT     20
```

Até aqui, entramos as instruções dos programas diretamente por números, como CDh, 21h. Mas isso é muito trabalhoso e, como veremos, há um modo bem mais fácil de entrar as instruções.

Além do comando de desassemblar, o Debug inclui um comando A (*Assemble — assemblar*), que permite que você entre com o mnemônico, ou instruções legíveis ao homem, diretamente. Assim, ao invés de entrar com esses números codificados para o nosso pequeno programa, podemos usar o comando de montagem para entrar o seguinte:

```
-A 100
3970:0100 INT 21
3970:0102 INT 20
3970:0104
```

Quando você tiver acabado de incluir as instruções, tudo que você tem a fazer é pressionar a tecla Enter, e o pronto do Debug reaparece.

Aqui, o comando A disse ao Debug que queríamos entrar com as instruções em forma mnemônica, e o 100 em nosso comando disse ao Debug para começar a entrada a partir da posição 100. Como o comando A do Debug torna a entrada bem mais simples, iremos usá-lo de agora em diante para entrar com as instruções.

MOVendo Dados Para os Registradores

Embora estivéssemos confiando muito no Debug até aqui, nem sempre iremos executar os programas com ele. Normalmente, um programa iria automaticamente setar os registradores AH e DL antes de uma instrução INT 21h. Para fazer isto, aprenderemos uma outra instrução: MOV. Quando soubermos bastante sobre esta instrução, seremos capazes de pegar um pequeno programa para imprimir um caractere e fazer um programa real — um que podemos executar diretamente do DOS.

Logo iremos usar a instrução MOV para colocar números nos registradores AH e DL. Mas vamos começar o estudo sobre MOV movendo números entre os próprios registradores. Coloque 1234h em AX (12h em AH, e 34h em AL), e ABCDh em DX (ABh em DH, e CDh em DL). Agora, entre a seguinte instrução com o comando A 100:

```
396F:0100 88D4        MOV       AH,DL
```

Esta instrução *move* o numero em DL, para AH, colocando uma copia dele em AH; AL não é afetado. Se você executar somente esta linha, descobrirá que AX = CD34 e que DX = ABCDh. Somente AH mudou. Agora ele contem uma cópia do número em DL.

Assim como o comando BASIC LET AH = DL, uma instrução MOV copia um número do segundo registrador para o primeiro, e por esta razão escrevemos AH antes de DL. Embora haja algumas restrições, que iremos verificar mais tarde, podemos usar outras formas de instrução MOV para copiar números entre outros pares de registradores. Por exemplo, resete IP e tente isto:

```
396F:0100 893C        MOV       BX,AX
```

Você acabou de mover palavras, e não bytes de registradores. A instrução MOV sempre atua entre palavras e palavras ou entre bytes e bytes; nunca entre palavras e bytes. Isso faz sentido, pois como moveria você uma palavra para um byte.

Inicialmente nós movemos um número para os registradores AH e DL. Vamos fazer isso de outra forma pela instrução MOV:

```
396F:0100 B402        MOV       AH,02
```

Esta instrução move 02 para o registrador AH sem afetar o conteúdo do registrador AL. O segundo byte da instrução, 02h é o número que desejamos mover. Tente mover um número diferente para AH: mude o segundo byte para outro valor, como C1h, com o comando E 101.

Agora, vamos juntar as partes deste capítulo e montar um programa maior. Ele irá imprimir um asterisco (*), por si mesmo, sem que precisemos ajustar os registradores (AH e DL). O programa usa instruções MOV para colocar os valores em AH e DL antes da chamada INT 21h ao DOS:

```
396F:0100 B402      MOV     AH,02
396F:0102 B22A      MOV     DL,2A
396F:0104 CD21      INT     21
396F:0106 CD20      INT     20
```

Entre com o programa e verifique-o com o comando U (U 100). Assegure-se de que IP aponta para a posição 100h, e então tente o comando G para executar o programa inteiro. Você deverá ver o caractere * aparecer na tela:

```
-G
*
Program terminated normally
```

Agora podemos ter um programa completo, vamos gravá-lo no disco como um programa .COM, para que possamos executá-lo diretamente do DOS. Podemos executar um programa .COM pelo DOS simplesmente digitando seu nome. Como nosso programa ainda não tem um nome, precisamos arrumar um.

O comando N (de *Nome*) do Debug dá um nome ao arquivo antes de gravá-lo no disco. Digite:

```
-N IMPLETRA.COM
```

para dar o nome IMPLETRA.COM ao nosso programa. Este comando, no entanto, ainda não grava nosso arquivo no disco — ele simplesmente nomeia o arquivo.

A seguir, devemos dar uma contagem de bytes ao Debug, informando o número de bytes em nosso programa, para que ele saiba quanta memória nós queremos gravar no arquivo. Se você olhar para a listagem do nosso programa, poderá ver que cada instrução ocupou dois bytes (isso não será sempre assim). Temos quatro instruções, e portanto nosso programa ocupa 4 * 2 = 8 bytes. (Também poderíamos usar o comando H do Debug, e usar a hexaritmética para determinar o número de bytes no programa. Digitando *H 108 100*, onde 108 é o endereço da instrução após INT 20, obteremos 8 como a diferença entre eles.)

Uma vez tendo o nosso número de bytes, precisamos de algum lugar para colocá-lo. O Debug usa o par de registradores BX:CX para o tamanho do nosso arquivo, de modo que colocar 8h em CX diz ao Debug que nosso programa possui oito bytes de comprimento. Finalmente, como nosso arquivo possui somente 8 bytes, devemos também zerar BX.

Definidos o nome e o tamanho do programa, podemos gravá-lo no disco como comando W (de *Write*) do Debug:

```
-W
Writing 0008 bytes
```

Agora temos um programa em nosso disco chamado IMPLETRA.COM. Logo, vamos sair do Debug, com um Q, e procurá-lo. Use o comando DIR do DOS para listar o arquivo:

```
A>DIR IMPLETRA.COM

 Volume in drive A has no label
Directory of A:\

IMPLETRA COM
        1 File(s)          18432 bytes free

A>
```

A lista do diretório nos diz que IMPLETRA.COM está no disco e que possui oito bytes, como deveria ser. Para executar o programa, digite simplesmente IMPLETRA após o pronto do DOS. Você deverá ver na tela o caractere indicado por meio do registrador DL no programa. E é só!

Imprimindo um String de Caracteres

Como exemplo final para este capítulo, usaremos INT 21h, com um número de função diferente no registrador AH, para escrever um grupo maior de caracteres. Teremos que colocar o nosso string de caracteres na memória e dizer ao DOS onde encontrar o string. Neste processo, aprenderemos um pouco mais sobre os endereços na memória.

Já vimos que a função de número 02h para INT 21h imprime um caractere na tela. Outra função, número 09h, imprime um string inteiro e pára de imprimir os caracteres quando encontrar um símbolo $ no string. Vamos colocar um string na memória. Começaremos na posição 200h, para que o string não se encontre com a parte do código do nosso programa. Entre com os seguintes números, usando a instrução E 200:

```
4F  6C  61  2C
20  44  4F  53
20  61  71  75
69  21  24
```

O último número, 24h, é o código ASCII para o sinal $, e ele diz ao DOS que é o final da seqüência de caracteres. Você verá o que diz esse string logo a seguir, quando executar o programa listado aqui:

```
396F:0100 B409          MOV     AH,09
396F:0102 BA0002        MOV     DX,0200
396F:0105 CD21          INT     21
396F:0107 CD20          INT     20
```

200h é o endereço do string que entramos, e colocar 200h em DX diz ao DOS onde encontrar o string de caracteres. Verifique o seu programa com o comando U, e depois execute-o com o comando G:

```
-G
Ola, DOS aqui!
Program terminated normally
```

Agora que armazenamos alguns caracteres na memória, é hora de aprendermos outro comando do Debug, D (de *Dump — cópia*). O comando de cópia apresenta a memória na tela de certo modo assim como U lista as instruções. Assim como você usou o comando U, simplesmente coloque um endereço após o comando D para dizer ao Debug onde começar a cópia. Por exemplo, digite o comando D 200 para ver uma cópia do string que você entrou:

```
-D 200
396F:0200  4F 6C 61 2C 20 44 4F 53-20 61 71 75 69 21 24 2E   Ola, DOS aqui!$.
396F:0210  24 5D C3 55 83 EC 30 8B-EC C7 06 10 00 00 00 E8   $]CU.10.IG.....h
```

Após cada par de endereços (tal como 396F:0200 em nosso exemplo), vemos 16 bytes hexa, seguidos pelos caracteres ASCII para estes bytes. Assim, na primeira linha você pode ver todos os caracteres ASCII que você digitou. O sinal de $ que você digitou é o penúltimo caractere da primeira linha; as outras informações não possuem significado para nós, foram deixadas por algum programa que usou essa região da memória.

Sempre que você vir um ponto (.) na parte de caracteres ASCII, ele representa ou um ponto realmente, ou um caractere especial, como a letra grega pi. O comando D do Debug apresenta somente 96 dos 256 caracteres no conjunto de caracteres ASCII do IBM PC, de modo que um ponto é usado para os outros 160 caracteres não apresentáveis.

Usaremos o comando D mais adiante para verificar os números que entramos como dados, sendo eles caracteres ou números mesmo. (Para maiores informações, veja a seção do Debug no seu manual do DOS).

Nosso programa para escrever string está completo, e podemos gravá-lo no disco. A rotina é a mesma que usamos para gravar IMPLETRA.COM no disco, mas desta vez temos que definir nosso tamanho de programa como tamanho suficiente para incluir o string em 200h. Nosso programa começa na linha 100, e podemos ver pela cópia da memória que o primeiro caractere após o $ está na posição 020F. Novamente, podemos usar o comando H para calcular a diferença entre esses dois números. Calcule 20Fh — 100h e armazene este valor no registrador CX, novamente zerando BX. Use o comando N para dar nome ao programa (adicione a extensão .COM para poder executar o programa diretamente pelo DOS), e depois use o comando W para gravar o programa e os dados para um arquivo de disco.

Isso serve para imprimir os caracteres na tela — mas há uma nota final: você pode ter notado que o DOS nunca emite um caractere $. Isso acontece assim porque o DOS usa o sinal $ para marcar o fim da seqüência de caracteres. Isso significa que não podemos usar o DOS dessa forma para imprimir um string com um caractere $, mas no capítulo seguinte veremos como imprimir o sinal $, ou qualquer outro caractere especial.

Resumo

Nossa preparação nos primeiros dois capítulos nos levou a um ponto em que pudemos escrever um programa real. Neste capítulo, usamos o conhecimento dos números hexa, Debug, instruções do 8088, e memória para montar pequenos programas para imprimir um caractere e uma série de caracteres na tela. Nesse processo, também aprendemos algumas coisas novas.

Primeiro aprendemos sobre as instruções INT — não em muitos detalhes, mas o suficiente para escrevermos dois pequenos programas. Nos capítulos seguintes, aprenderemos mais sobre as interrupções conforme aumentamos nosso conhecimento sobre o microprocessador 8088 que está encoberto pelo seu IBM PC.

O Debug, mais uma vez, tem sido um guia útil e fiel. Usamos bastante o Debug para apresentar o conteúdo dos registradores e da memória, e neste capítulo usamos ainda mais as suas capacidades. O Debug executou nossos programas com o comando G.

Também aprendemos sobre a instrução de saída INT 20, e a instrução MOV, que move os números para e entre os registradores. A instrução de saída (INT 20) nos permitiu criar um programa completo que pudesse ser gravado em disco e executado diretamente pelo DOS, sem a ajuda do Debug. E a instrução MOV nos permitiu setar os registradores antes de uma instrução INT 21 (imprimir), de modo que pudemos escrever um programa completo para imprimir um caractere qualquer.

Finalmente, fechamos o capítulo com a função INT 21h para imprimir uma série inteira de caracteres. Usaremos bastante todas estas instruções no restante deste livro, mas como você viu pelo uso dos comandos A e U do Debug, você não precisará lembrar dos códigos de máquina para estas instruções.

Agora já sabemos o bastante para poder imprimir números binários. No capítulo a seguir, faremos um pequeno programa para tomar um byte e imprimi-lo na tela como um string de dígitos binários (zeros e uns).

4

IMPRIMINDO NÚMEROS BINÁRIOS

Rotações e o Flag de Carry

Somando com o Flag de Carry

Loops

Imprimindo um Número Binário

O Comando de Prosseguimento

Resumo

Neste capítulo, montaremos um programa para escrever números binários na tela como seqüências de zeros e uns. Já sabemos quase tudo que precisamos, e nosso trabalho aqui ajudará a solidificar as idéias já apresentadas. Também aprenderemos algumas instruções adicionais, incluindo outra versão de ADD e algumas instruções para nos ajudarem a repetir partes do nosso programa. Vamos começar aprendendo algo completamente novo.

Rotações e o Flag de Carry

No Capítulo 2, quando vimos inicialmente a aritmética hexa, descobrimos que, ao somar 1 a FFFFh, deveríamos ter 10000h, mas não. Somente os quatro dígitos hexa à direita cabem em uma palavra; o 1 não. Também descobrimos que este 1 é um estouro (overflow) e que não é perdido. Para onde ele vai? Ele é colocado em algum lugar chamado *flag* — neste caso, o *Flag de Carry*, ou *CF*. Os flags contêm números de 1 bit, de modo que podem conter ou zero ou um. Se precisarmos passar um para o quinto dígito hexa, ele vai para o flag de carry.

Voltemos a nossa instrução ADD do Capítulo 3 (ADD AX,BX). Coloque FFFFh em AX e 1 em BX, e depois execute a instrução ADD. Ao final da segunda linha da apresentação do comando R do Debug, você verá oito pares de letras. O último deles, que pode ser NC ou CY, é o flag de carry. Nesse momento, por ter a instrução de soma causado um estouro de 1, você verá que o status do carry é CY (*Carry*). O bit de carry agora é 1 ou, como diremos, ele está *setado*.

Só para confirmar que armazenamos o décimo sétimo bit aqui (ele seria o nono bit em uma adição de bytes), some 1 ao zero em AX resetando IP para 100h e executando a instrução novamente com T. O flag de carry é afetado por cada instrução de adição, e desta vez não deve haver qualquer estouro, de modo que o carry será resetado. E, na verdade, o carry torna-se zero, conforme indicado pelo NC, que significa *Nenhum Carry*, como pode ser visto pelo comando R.

(Veremos mais sobre os outros flags de status depois; mas, se você quer saber agora, terá mais informações sobre eles através do comando R do Debug no seu manual do DOS.)

Vamos examinar a tarefa de imprimir um número binário, para ver como a informação do carry pode ser útil. Devemos imprimir somente um caractere de cada vez, e queremos apanhar os bits do número, um a um, da esquerda para a direita. Por exemplo, o primeiro caractere que iríamos apanhar para imprimir o número 1000 0000b seria o um. Se pudéssemos mover este byte inteiro para a esquerda em uma casa, colocar o um no flag de carry, e acrescentar um zero no lado direito, e depois repetir o processo para cada dígito subseqüente, o flag de carry conteria nossos dígitos binários. E podemos fazer exatamente isto com uma nova instrução chamada RCL (*Rotate Carry Left — Rotação à Esquerda do Flag*).

Para ver como isso funciona, entre o pequeno programa:

```
3985:0100 D0D3          RCL     BL,1
```

Esta instrução roda o byte em BL para a esquerda em um bit (daí o 1 na instrução), e faz isso por meio do flag de carry. A instrução é chamada de rotação, pois RCL move o bit da esquerda para o flag, enquanto move o bit anteriormente no flag de carry para a posição do bit à direita (bit 0). No processo, todos os outros bits são movidos, ou rodados, para a esquerda. Após algumas rotações (17 para uma palavra, e 9 para um byte) os bits são movidos de volta para suas posições originais, e você está de volta à posição original. Pense nesta instrução como um jogo de vôleibol. Sempre que um dos times obtém uma vantagem, todos os jogadores rodam em uma posição, e o jogador do canto direito, longe da rede, passa para o saque.

Coloque B7h no registrador BX, e depois execute esta instrução diversas vezes. Convertendo seus resultados para binário, você verá o seguinte:

Carry	Registrador BL			
0	1011	0111	B7h	Iniciamos aqui
1	0110	1110	6Eh	
0	1101	1101	DDh	
1	1011	1010	BAh	
0	1011	0111	B7h	Após 9 rotações

Na primeira rotação, o bit 7 de BL move-se para o flag de carry, o bit no flag de carry move-se para o bit 0 de BL, e todos os outros bits movem-se uma posição à esquerda até que, após nove rotações, o número original está de volta ao registrador BL.

Estamos próximos de conseguir o nosso programa para escrever números binários na tela, mas ainda precisamos de algumas outras partes. Vejamos como podemos converter o bit no flag de carry para o caractere 0 ou 1.

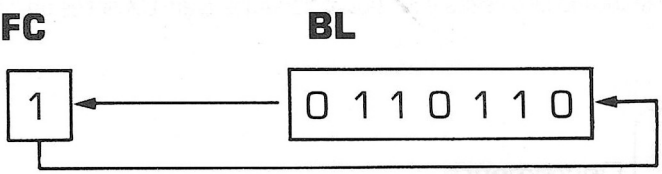


Figura 4-1. ***A Instrução RCL BL,1.***

Somando com o Flag de Carry

A instrução ADD normal, por exemplo ADD AX,BX, simplesmente soma dois números. Uma outra instrução, ADC (*Adição com Carry*) soma três números: os dois, como antes, *mais* o bit do flag de carry. Se você olhar em sua tabela ASCII, descobrirá que 30h é o caractere 0, e 31h é o caractere 1. Assim, a soma do flag de carry com 30h nos dá o caractere 0 quando o flag estiver zerado e 1 quando ele estiver setado. Assim, se DL = 0 e o flag de carry estiver setado, a execução de:

```
ADC     DL,30
```

somará DL (0) a 30h ("0"), e com o flag setado (1), DL conterá 31h ("1"). Logo, com apenas uma instrução, conseguimos converter o carry para um caractere que pode ser impresso.

Neste ponto, ao invés de fazer um exemplo com ADC, vamos esperar por nosso programa completo. Quando tivermos montado o nosso programa, executaremos suas instruções uma de cada vez, numa tarefa *passo-a-passo*, e através disto, veremos tanto como funciona a instrução ADC como também o modo com que ela se encaixa bem ao nosso programa. Mas primeiro precisamos de mais uma instrução, que usaremos para repetir nossas instruções RCL, ADC e INT 21h (imprimir) oito vezes: uma para cada bit do byte.

Loops

Como dissemos, a instrução RCL não se limita a rodar os bits; ela também roda palavras inteiras. Vamos usar esta capacidade para demonstrar a instrução *LOOP*. LOOP é algo como um ciclo FOR-NEXT do BASIC, mas não de um modo geral. No entanto, assim como o FOR-NEXT do BASIC, precisamos dizer ao LOOP quantas vezes ele deve ser executado. Fazemos isto colocando nosso contador de repetições no registrador CX. Toda vez que o ciclo for executado, o 8088 subtrai um de CX, e, quando CX se tornar zero, LOOP termina o ciclo.

Por que o registrador CX? O C em CX significa *Contador*. Podemos usar este registrador para qualquer finalidade, mas, como você verá no capítulo seguinte, o registrador CX é usado com outras instruções quando queremos repetir operações. Eis aqui um programa simples que roda o registrador BX oito vezes, movendo BL para BH (mas não o contrário, pois rodamos através do flag de carry):

```
396F:0100 BBC5A3    MOV     BX,A3C5
396F:0103 B90800    MOV     CX,0008
396F:0106 D1D3      RCL     BX,1
396F:0108 E2FC      LOOP    0106
396F:010A CD20      INT     20
```

Nosso ciclo começa em 106h (RCL BX,1) e termina com a instrução LOOP. O número após LOOP (106h) é o endereço da instrução RCL. Quando executarmos o programa, LOOP subtrai um de CX, e depois salta para o endereço 106h se CX não for zero. A instrução RCL BX,1 (roda o flag de carry à esquerda em uma posição) é executada oito vezes aqui, pois colocamos 8 em CX antes do ciclo se iniciar.

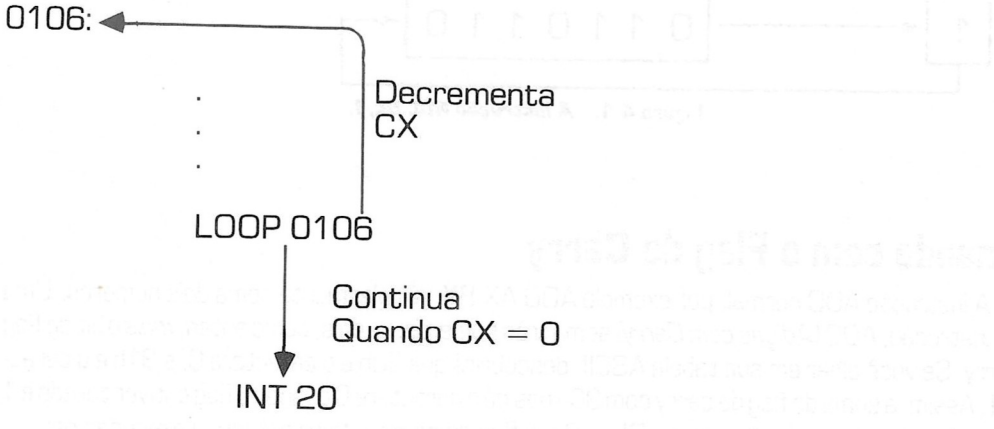


Figura 4-2. ***A Instrução LOOP.***

Você deve ter notado que, diferente do FOR-NEXT do BASIC, a instrução LOOP está no final do ciclo (onde teríamos colocado o NEXT no BASIC). E no início do ciclo, a instrução RCL em 106h, não há uma instrução especial como o FOR no BASIC. Se você conhece uma linguagem como o Pascal, você pode notar que a instrução LOOP é algo mais parecido com o par de instruções REPEAT-UNTIL, onde a instrução REPEAT só indica o início do bloco de instruções que ele deve saltar.

Há diferentes modos de executarmos nosso pequeno programa. Se você simplesmente digitar G, não verá qualquer mudança nos registradores, pois o Debug salva todos os registradores antes de começar a executar o comando G. Depois, se ele encontrar uma instrução INT 20 (como acontecerá em nosso programa), ele recupera todos os registradores. Tente G. Você verá que IP foi resetado para 100h (onde você começou) e que os outros registradores também não são diferentes.

Se você tiver paciência, poderá executar este programa, com uma instrução de cada vez. Assim, você poderá ver os registradores mudarem a cada passo:

```
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100   NV UP DI PL NZ NA PO NC
0CDE:0100 BBC5A3          MOV         BX,A3C5
-T
```

```
AX=0000 BX=A3C5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0103   NV UP DI PL NZ NA PO NC
0CDE:0103 B90800          MOV         CX,0008
-T

AX=0000 BX=A3C5 CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106   NV UP DI PL NZ NA PO NC
0CDE:0106 D1D3            RCL         BX,1
-T

AX=0000 BX=478A CX=0008 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108   OV UP DI PL NZ NA PO CY
0CDE:0108 E2FC            LOOP        0106
-T

AX=0000 BX=478A CX=0007 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0106   OV UP DI PL NZ NA PO CY
0CDE:0106 D1D3            RCL         BX,1
-T


-T

AX=0000 BX=C551 CX=0001 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0108   NV UP DI PL NZ NA PO CY
0CDE:0108 E2FC            LOOP        0106
-T

AX=0000 BX=C551 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=010A   NV UP DI PL NZ NA PO CY
0CDE:010A CD20            INT         20
-T
```

Alternadamente, você poderá digitar G 10A para executar o programa até, mas não incluindo, a instrução INT 20 em 10Ah; depois os registradores mostrarão o resultado do programa.

Se você tentar isto, verá CX = 0 e ou BX = C551 ou BX = C5D1, dependendo do valor do carry antes da execução do programa. O C5 que a instrução MOV do programa colocou em BL logo no início está agora no registrador BH, mas BL não contém A3, pois rodamos BX *por dentro* do carry. Mais tarde, veremos outros modos de rodar os registradores sem que passemos pelo carry. Vamos voltar ao nosso objetivo, que é imprimir um número em sua forma binária.

Imprimindo um Número Binário

Já vimos como separar os dígitos binários um de cada vez, e convertê-los para caracteres ASCII. Se adicionarmos uma instrução INT 21h para imprimir nossos dígitos, nosso programa estará pronto. Olhe só para ele; a primeira instrução coloca 02 em AH para a chamada de função INT 21h (lembre-se, 02 diz ao DOS para imprimir o caractere no registrador DL):

```
3985:0100 B402            MOV         AH,02
3985:0102 B90800          MOV         CX,0008
3985:0105 B200            MOV         DL,00
3985:0107 D0D3            RCL         BL,1
3985:0109 80D230          ADC         DL,30
3985:010C CD21            INT         21
3985:010E E2F5            LOOP        0105
3985:0110 CD20            INT         20
```

Já vimos como todas as partes trabalham, e agora as colocamos todas juntas. Use a rotação de BL (com a instrução RCL BL,1) para selecionar os bits de um número, apanhe o número que você quer imprimir em binário, carregue-o no registrador BL, e depois execute este programa com o comando G. Após a instrução INT 20h, o comando recupera os registradores para os valores que eles tinham antes, de modo que BL ainda contém o número que você imprimiu em binário.

A instrução ADC DL,30 em nosso programa converte o flag de carry para um caractere zero ou um. A instrução MOV DL,0 zera o registrador DL, e depois a instrução ADC soma 30h a DL, e então finalmente soma o carry. Como 30h é o código ASCII para um 0, o resultado de ADC DL,30 é o código ASCII para 0 quando o flag de carry estiver zerado (NC) ou 1 se o carry estiver setado (CY).

Se você quiser ver o que acontece enquanto este programa estiver sendo executado, acompanhe-o passo a passo. Tenha em mente que você deverá ter um pouco de cuidado quando usar o comando T. O programa contém uma instrução INT 21h e, como você viu quando inicialmente encontramos INT 21h, o DOS realiza muito trabalho para essa única instrução. E é por isso que você não deve usar T onde houver INT 21.

Você pode, no entanto, acompanhar todas as outras instruções neste programa, com exceção do INT 20 final, que não interessa enquanto não for o final do programa. Durante o seu trabalho, toda vez que você atingir a instrução INT 21h, digite G10E. O seu comando G, seguido por um endereço, dirá ao Debug para continuar a executar o programa, mas para parar quando IP se tornar o endereço que você entrou (10E). Isto é, o Debug executará a instrução INT 21h sem que você tenha que passar por todos os seus passos, mas para antes de executar a instrução em 10E, para que você possa continuar a acompanhar o programa passo a passo. (O número que você digita após G é conhecido como *"breakpoint"* (*ponto de parada*) no manual do DOS; esses pontos são muito úteis quando você está tentando entender o funcionamento interno dos programas.)

Finalmente, termine o programa quando você atingir a instrução INT 20h, digitando o comando G isolado.

O Comando de Prosseguimento

Tendo ou não tentado as instruções de acompanhamento dos passos de nosso programa, você viu que uma instrução como G 10E nos permite *pular* uma instrução como INT, que começa em, digamos, 10Ch. Mas isso significa que toda vez que quisermos pular uma instrução INT, nós temos que descobrir o endereço da instrução que vem logo após a instrução INT.

Para isso, há um comando do Debug que torna bem mais simples o acompanhamento das instruções. O comando P (de Prosseguir) faz todo o trabalho para nós. Para ver isso, execute o programa passo a passo, mas, desta vez, quando você chegar à instrução INT 21h, digite P, ao invés de G 10E, conforme explicamos anteriormente.

Usaremos bastante o comando P no decorrer deste livro, pois é um modo muito fácil de saltarmos instruções como INT, que chamam grandes programas, tais como as rotinas dentro do DOS. Mas, antes de continuarmos, devemos falar algo sobre o comando P — ele não está documentado nas versões do DOS anteriores à 3.00. Esta falta na documentação pode ter sido por esquecimento ou, mais provavelmente, porque a Microsoft não teve tempo de testar o comando P totalmente antes de entregar a versão 2.00 do DOS. Qualquer que seja a razão, se você possui uma versão do DOS anterior à 3.00, deve estar avisado que o comando P *pode não* funcionar corretamente todas as vezes — embora nunca tenhamos tido problemas com o seu uso.

Isto é tudo o que faremos para imprimir números binários como uma seqüência de zeros e uns, mas tente fazer um simples exercício: Veja se você pode modificar este programa para imprimir um *b* no final do nosso número binário. (**Dica:** O código ASCII de "b" é 62h).

Resumo

Neste capítulo, tivemos a chance de realizar algo útil após todos os novos conceitos apresentados nos Capítulos de 1 a 3. Mas até onde nós fomos, e o que vimos?

Tivemos o nosso primeiro encontro com os flags, e demos uma olhada no flag de carry, que foi de interesse especial aqui, pois tornou nossa tarefa de imprimir um número em binário muito mais simples. Isso foi possível após aprendermos sobre a instrução de rotação RCL, que gira um byte ou palavra à esquerda, um bit de cada vez.

Quando tínhamos aprendido sobre o flag de carry e a rotação de bytes e palavras, apareceu uma nova versão da instrução de adição, ADC, e estávamos quase prontos para montarmos o nosso programa que imprime um número em notação binária.

Foi aí que a instrução LOOP entrou em cena. Carregando o registrador CX com um contador de ciclos (ou loops), pudemos manter o 8088 executando algumas instruções por uma certa quantidade de vezes. Definimos CX como 8, para que o loop fosse executado oito vezes.

Foi só isso que precisamos saber para escrever nosso programa. Usaremos estas ferramentas novamente nos capítulos seguintes. No próximo capítulo, imprimiremos um número binário em notação hexadecimal, assim como o Debug faz, de modo que quando terminarmos o Capítulo 5, teremos uma idéia melhor de como o Debug traduz os números do binário para o hexa. Depois, passaremos para o outro extremo do Debug: a leitura de números digitados em hexa, e a conversão deles para a notação binária do 8088.

5

IMPRIMINDO EM HEXA

Bits de Comparação e Status

Imprimindo um Único Dígito Hexa

Outra Instrução de Rotação

Lógica e AND

Juntando as Partes

Resumo

Nosso programa do Capítulo 4 foi bem direto. Tivemos sorte porque o flag de carry tornou fácil imprimir um número binário como um string de caracteres 0 e 1. Agora iremos imprimir números em notação hexa. Aqui, nosso trabalho será um pouco menos direto, e começaremos a repetir dentro dos programas, escrevendo a mesma seqüência de instruções mais de uma vez. Mas esse tipo de repetição não durará para sempre: no Capítulo 7, aprenderemos sobre as procedures, ou sub-rotinas, que eliminam a necessidade de se escrever mais de uma cópia de um grupo definido de instruções. Primeiro, vamos aprender sobre um grupo de instruções e ver como imprimir os números em hexa.

Bits de Comparação e Status

No último capítulo, aprendemos algo sobre os flags de status e examinamos especialmente o flag de carry, que é representado ou por CY ou por NC no comando R do Debug. Os outros flags, igualmente úteis, cuidam do *status* da última operação aritmética. Há oito flags, e iremos aprender sobre eles quando for necessário.

Lembre-se de que CY significa que o flag de carry é igual a 1, ou que está setado, enquanto NC significa que o flag de carry é zero. Em todos os flags, 1 significa *verdadeiro* e 0 significa *falso*. Por exemplo, se você fez uma instrução SUB com um resultado de 0, o flag conhecido como Flag de Zero será 1 — verdadeiro — e você o verá na linha de flags do comando R como ZR (*Zero*). Caso contrário, o flag de zero seria resetado em 0 — NZ (*Não Zero*).

Vejamos um exemplo que testa o flag de zero. Usaremos a instrução SUB para subtrair dois números. Se os dois números forem iguais, o resultado será zero, e o flag de zero aparecerá como ZR na tela. Entre com a seguinte instrução de subtração:

```
396F:0100 29D8          SUB     AX,BX
```

Agora, execute esta instrução com alguns números diferentes, verificando se aparece ZR ou NZ no flag de zero. Se você colocar o mesmo número nos dois registradores, AX e BX, você verá o flag de zero setado após uma instrução de subtração, e zerado após a outra:

```
-R
AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100   NV UP DI PL NZ NA PO NC
0CDE:0100 29D8          SUB     AX,BX
-T

AX=0000 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102   NV UP DI PL ZR NA PE NC
0CDE:0100 3F            AAS
-R IP
IP 0102
:100
-R
AX=0000 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0100   NV UP DI PL ZR NA PE NC
0CDE:0100 29D8          SUB     AX,BX
-T

AX=FF0B BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102   NV UP DI NG NZ AC PO CY
0CDE:0102 3F            AAS
```

Se subtrairmos um de zero, o resultado será FFFFh, que, como dissemos no Capítulo 1, é −1, a na forma de complemento de dois. Será que podemos dizer pela apresentação de R se um número é negativo ou positivo? Sim; outro flag, chamado *Flag de Sinal*, alterna-se entre NG (*Negativo*) e PL (*Plus — Mais*), e é setado para 1 quando um número é um negativo em complemento de dois.

Ainda outro flag que nos interessa é o Flag de Overflow, que alterna-se entre OV (*Overflow*) quando o flag for 1, e NV (*Não Overflow*) quando o flag for zero. O flag de overflow é setado se o bit de sinal mudar quando não deveria. Por exemplo, se somarmos dois números positivos, como 7000h e 6000h, obtemos um número negativo, D000h, ou −12288. Isto é um erro porque o resultado ultrapassa o tamanho da palavra (houve um overflow — estouro). O resultado deveria ser positivo, mas não é, e por isso o 8088 seta o flag de overflow. (Lembre-se, se estivéssemos lidando com números sem sinal, isto não seria um erro, e ignoraríamos o flag de overflow.)

Tente diversos números diferentes para ver se você pode setar e resetar cada um destes flags, experimentando-os até que esteja familiarizado com eles. Para o overflow, subtraia um número negativo grande de um número positivo grande — por exemplo, 7000h — 8000h, visto que 8000h é um número negativo igual a −32768 na forma de complemento de dois.

Agora estamos prontos para ver um grupo de instruções chamadas instruções de *salto condicional*. Elas nos permitem checar os flags de status de forma mais conveniente do que fizemos até aqui. A instrução JZ (*Jump if Zero — salta se zero*) salta para um novo endereço se o último resultado aritmético foi zero. Assim, se seguirmos uma instrução SUB com, por exemplo, JZ 15A, um resultado zero para a subtração causaria um salto do 8088 (e o começo da execução) para os comandos no endereço 15A, ao invés da próxima instrução.

A instrução JZ testa o flag de zero, e, se estiver setado (ZR), faz o salto da mesma forma que um salto com o comando do BASIC IF A = 0 THEN 100. O oposto de JZ é JNZ (*Jump if Not Zero — salta se não for zero*). Vejamos um exemplo simples que usa JNZ e subtrai um de um número até que o resultado seja zero:

```
396F:0100 2C01        SUB     AL,01
396F:0102 75FC        JNZ     0100
396F:0104 CD20        INT     20
```

Coloque um número (digamos, 3) em AL, para que você passe pelo loop poucas vezes, e depois execute este programa passo a passo, uma instrução de cada vez, para ver como funcionam os saltos condicionais. Colocamos a instrução INT 20h no final para que, se você incidentalmente digitar G, o programa não siga adiante: essa é uma boa prática de defesa.

Você deve ter notado que o uso de SUB para comparar dois números, como fizemos aqui, tem o efeito indesejado de alterar o primeiro número. Isto é, se AL continha 3, ao final ele conterá 0, alterando o valor original. Uma outra instrução, CMP (*Compare*) nos permite fazer a subtração sem armazenar o resultado em qualquer lugar e sem alterar o primeiro número. O resultado é usado somente para setar os flags, de modo que podemos usar uma das muitas instruções de salto condicional após uma comparação. Para ver o que acontece, coloque o mesmo número em AX e BX, F5h, e acompanhe esta instrução:

```
-A 100
0CDE:0100 CMP AX,BX
0CDE:0102
-T

AX=00F5 BX=00F5 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0CDE ES=0CDE SS=0CDE CS=0CDE IP=0102   NV UP DI PL ZR NA PE NC
0CDE:0102 3F            AAS
```

O flag de zero agora está setado (ZR), mas F5 permanece em ambos os registradores.

Vamos usar CMP para imprimir um único dígito hexa. Criaremos um grupo de instruções que use os flags para alterar o fluxo do programa, assim como LOOP fez no último capítulo, de forma semelhante ao comando IF-THEN do BASIC. Este novo grupo de instruções usará os flags para testar condições como menor que, maior que, e assim por diante. Não teremos que nos preocupar com os flags que deverão estar setados quando o primeiro número for menor que o segundo; as instruções já sabem em que flag elas devem procurar.

Imprimindo um Único Dígito Hexa

Vamos começar colocando um pequeno número (entre 0 e Fh) no registrador BL. Como qualquer número entre 0 e F é equivalente a um dígito hexa, podemos converter nossa escolha para um único caractere ASCII e imprimi-lo. Vejamos os passos que precisamos para realizar esta conversão.

Caractere	Código ASCII (Hexa)
/	2F
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39
:	3A
;	3B
<	3C
=	3D
>	3E
?	3F
@	40
A	41
B	42
C	43
D	44
E	45
F	46
G	47

Figura 5-1. ***Tabela ASCII Parcial Mostrando os Caracteres Usados Pelos Dígitos Hexa.***

Os caracteres ASCII de 0 a 9 possuem os valores de 30h a 39h; os caracteres de A a F, contudo, possuem os valores de 41h a 46h. Aqui está o problema: estes dois grupos de caracteres ASCII estão separados por sete caracteres. Como resultado, a conversão para ASCII será diferente para os dois grupos de números (de 0 a 9 e de Ah a Fh). Portanto, devemos lidar com cada grupo de forma diferente. Um programa em BASIC para fazer esta conversão em duas partes seria mais ou menos assim:

```
100 IF  BL < &H0A
      THEN BL = BL + &H30
      ELSE BL = BL + &H37
```

(Note que escrevemos 0A para o número A, ao invés de A puro, para que não confundamos o número Ah com o registrador AH. Geralmente colocaremos um dígito 0 antes dos números em hexa para

tais situações, que podem causar confusão. Na verdade, como não custa nada colocarmos um zero antes de um número hexa, é uma boa idéia colocar um zero antes de *todos* os números hexa.)

Nosso programa de conversão em BASIC é muito simples. Infelizmente, a linguagem de máquina do 8088 não possui um comando ELSE; ela é muito mais primitiva do que a linguagem BASIC, de modo que precisaremos de algo mais inteligente. Eis outro programa em BASIC, desta vez imitando o método que usaremos para o nosso programa em linguagem de máquina:

```
100 BL = BL + &H30
110 IF BL >= &H3A
      THEN BL = BL + &H7
```

Você pode convencer-se de que este programa funciona executando-o com alguns exemplos. Os números 0,9,Ah e Fh são particularmente bons porque estes quatro cobrem todas as condições de *limite* — áreas onde geralmente começam os problemas.

Aqui, 0 e Fh são, respectivamente, o menor e o maior número hexa de um dígito, e usando 0 e Fh podemos checar o início e o fim de nossa faixa. Os números 9 e 0Ah, embora próximos um do outro, precisam de dois esquemas de conversão diferentes em nosso programa. Usando 9 e 0Ah, confirmamos que foi escolhido o lugar correto para se passar de um esquema de conversão para outro.

A versão em linguagem de máquina deste programa contém alguns poucos passos a mais, mas é essencialmente o mesmo que a versão em BASIC. Ele usa a instrução CMP, assim como uma instrução de salto condicional chamada JL (*Jump if Less Than — salta se menor que*). Eis aqui o programa para apanhar um único dígito hexa em BL e imprimi-lo em hexa:

```
3985:0100 B402      MOV     AH,02
3985:88DA           MOV     DL,BL
3985:0104 80C230    ADD     DL,30
3985:0107 80FA3A    CMP     DL,3A
3985:010A 7C03      JL      010F
3985:010C 80C207    ADD     DL,07
3985:010F CD21      INT     21
3985:0111 CD20      INT     20
```

A instrução CMP, como vimos antes, subtrai dois números (DL — 3Ah) para setar os flags, mas não altera o valor em DL. Assim, se DL for menor que 3Ah, a instrução JL 010F salta para a instrução INT 21 em 10Fh. Coloque um único dígito hexa em BL e execute este nosso exemplo para sentir um pouco melhor o uso de CMP e do nosso algoritmo para converter hexa para ASCII. Lembre-se de usar ou o comando G com um "breakpoint" ou o comando P quando atingir as instruções INT.

Figura 5-2. ***A Instrução JL.***

Outra Instrução de Rotação

Nosso programa funciona para qualquer número hexa de um dígito, mas se quisermos imprimir um número hexa de dois dígitos, precisamos de mais alguns passos. Precisamos isolar cada dígito (quatro bits, que são geralmente chamados de *nibble*) deste número hexa de dois dígitos. Nesta

seção, veremos que é fácil isolar os primeiros quatro bits, ou bits mais altos, e na seção seguinte, encontraremos um conceito conhecido por *operação lógica*, que usaremos para isolar os quatro bits mais baixos — o segundo dos nossos dois dígitos hexa.

Para começar, lembre-se que a instrução RCL roda um byte ou uma palavra à esquerda, através do flag de carry. No capítulo anterior nós usamos a instrução RCL BL,1, em que o 1 disse ao 8088 para rodar BL em somente 1 bit. Podemos rodar mais do que um bit, se quisermos, mas não podemos simplesmente escrever a instrução RCL BL,2. (**Nota:** Embora RCL BL,2 não seja uma instrução válida do 8088, ela trabalha corretamente com o processador 80286 encontrado nos IBM ATs. Mas como os IBM PCs antigos são mais comuns do que os ATs, é melhor que você escreva seus programas para o denominador comum — o velho 8088.) Para rotações com mais de um bit, devemos colocar um contador no registrador CL.

O registrador CL é usado aqui da mesma forma que o registrador CX é usado pela instrução LOOP para determinar o número de vezes em que o loop será repetido. Use CL para o número de vezes em que o byte ou palavra deve rodar, ao invés de CX, pois não faz sentido rodar mais do que 16 vezes; assim, o registrador de oito bits CL é mais do que suficiente para conter a contagem máxima de deslocamentos.

O que tudo isso tem a ver com a impressão de números hexa? Nosso plano agora é girar o byte em DL quatro bits à direita. Para fazer isso, usaremos uma instrução de rotação ligeiramente diferente, chamada SHR (*Shift Right — deslocamento à direita*). Usando SHR, podemos mover os quatro bits altos de nosso número para o nibble (quatro bits) da direita.

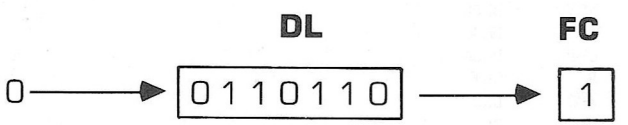


Figura 5-3. ***A Instrução SHR DL, 1.***

Também queremos zerar os quatro bits altos, de modo que o registrador inteiro se torne igual ao byte que estamos deslocando para a direita. Se fôssemos entrar com SHR DL,1, nossa instrução moveria o byte em DL um bit para a direita e, ao *mesmo* tempo, moveria o bit 0 para o flag de carry, quanto inclui um 0 no bit 7 (o mais alto, ou o da esquerda de DL). Se fizermos isso mais três vezes, teremos exatamente o que queremos: os quatro bits mais altos serão deslocados para os quatro bits baixos, enquanto as quatro posições altas serão completadas com zeros. Podemos fazer todo esse deslocamento com somente uma instrução, usando o registrador CL como *contador de deslocamento*. Setando CL para quatro antes da instrução SHR DL,CL, DL acabará tendo o dígito hexa superior.

Vejamos como tudo isso funciona. Coloque 4 em CL e 4Dh em DL e depois entre e execute a seguinte instrução SHR:

```
3985:0100 D2EA          SHR     DL,CL
```

DL agora deve ser 05h, que é o primeiro dígito do número 5Dh, e podemos agora imprimir este dígito com um programa como aquele que usamos anteriormente. Assim, juntando as partes que temos até aqui, podemos montar o seguinte programa para tomar um número no registrador BL e imprimir o primeiro dígito hexa:

```
3985:0100 B402          MOV     AH,02
3985:0102 88DA          MOV     DL,BL
3985:0104 B104          MOV     CL,04
3985:0106 D2EA          SHR     DL,CL
3985:0108 80C230        ADD     DL,30
3985:010B 80FA3A        CMP     DL,3A
3985:010E 7C03          JL      0113
3985:0110 80C207        ADD     DL,07
3985:0113 CD21          INT     21
3985:0115 CD20          INT     20
```

Lógica e AND

Agora que podemos imprimir o primeiro dos dois dígitos hexa de um byte, vejamos como podemos isolar e imprimir o segundo dígito. Dessa vez, aprenderemos como zerar os quatro bits superiores do nosso número original (não deslocado), deixando DL igual aos quatro bits menores. Muito simples: Zere os quatro bits altos com uma instrução chamada AND. A instrução AND é uma das importantes instruções chamadas *lógicas* — aquelas que possuem suas raízes na lógica formal. Vejamos como AND funciona.

Na lógica formal, podemos dizer: "A é verdadeiro se B e C forem ambos verdadeiros." Mas se B ou C for falso, então A deve também ser falso. Se tomarmos esta sentença, substituirmos "verdadeiro" por 1 e "falso" por 0, e depois olharmos as várias combinações possíveis de A, B e C, poderemos criar o que é conhecido como tabela *verdade*. Eis uma tabela verdade para a operação AND entre dois bits:

AND	F	V
F	F	F
V	F	V

=

AND	0	1
0	0	0
1	0	1

Na horizontal e na vertical estão os valores para os dois bits. Os resultados para o AND estão na tabela, de modo que você pode ver que 0 AND 1 dá 0.

A instrução AND trabalha com bytes e palavras comparando os bits dos bytes e das palavras que estão na mesma posição. Por exemplo, o comando AND BL,CL realiza operações AND sucessivas dos bits 0 de BL e CL, dos bits 1, bits 2, e assim por diante, e coloca o resultado em BL. Vamos esclarecer melhor isso com um exemplo direto em binário:

```
      0101    0101
AND   0111    0110
-------------------
0011    0100
```

Além disso, se realizarmos um AND de 0Fh com qualquer número, podemos zerar os quatro bits de qualquer números, sem afetar os quatro bits baixos:

```
      0111    1011
AND   0000    1111
-------------------
0000    1011
```

Vamos colocar esta lógica em um pequeno programa para tomar o número em BL, isolar o dígito hexa mais baixo com AND 0Fh, e depois imprimir o resultado como um caractere. Já vimos a maioria dos detalhes deste programa quando imprimimos o dígito mais alto; o único detalhe novo é a instrução AND:

```
3985:0100 B402          MOV     AH,02
3985:0102 88DA          MOV     DL,BL
3985:0104 80E20F        AND     DL,0F
3985:0107 80C230        ADD     DL,30
3985:010A 80FA3A        CMP     DL,3A
3985:010D 7C03          JL      0112
3985:010F 80C207        ADD     DL,07
3985:0112 CD21          INT     21
3985:0114 CD20          INT     20
```

Tente isto com alguns números hexa de dois dígitos em BL antes de juntarmos tudo para imprimir ambos os dígitos. Você deverá ver o dígito hexa da direita do seu número em BL impresso na tela.

Juntando as Partes

Na verdade, não há muito o que mudar quando juntarmos as duas partes. Precisamos somente alterar o endereço da segunda instrução JL que usamos para imprimir o segundo dígito hexa. Aqui esta o programa completo:

```
3985:0100 B402          MOV     AH,02
3985:0102 88DA          MOV     DL,BL
3985:0104 B104          MOV     CL,04
3985:0106 D2EA          SHR     DL,CL
3985:0108 80C230        ADD     DL,30
3985:010B 80FA3A        CMP     DL,3A
3985:010E 7C03          JL      0113
3985:0110 80C207        ADD     DL,07
3985:0113 CD21          INT     21
3985:0115 88DA          MOV     DL,BL
3985:0117 80E20F        AND     DL,0F
3985:011A 80C230        ADD     DL,30
3985:011D 80FA3A        CMP     DL,3A
3985:0120 7C03          JL      0125
3985:0122 80C207        ADD     DL,07
3985:0125 CD21          INT     21
3985:0127 CD20          INT     20
```

Com o programa incluído, você terá que digitar U 100, seguido de U, para ver a listagem inteira desassemblada. Note que nós repetimos um grupo de cinco instruções: as instruções de 108h a 113h, também presentes em 11Ah a 125h. No Capítulo 7 veremos como escrever esta seqüência de instruções somente uma vez, usando uma instrução semelhante ao comando GOSUB do BASIC.

Resumo

Neste capítulo, aprendemos mais sobre como o Debug traduz os números do formato binário do 8088 para um formato hexa que podemos ler facilmente. O que foi que acrescentamos a nossa massa cinzenta dessa vez?

Primeiro, aprendemos sobre alguns dos flags de duas letras que vemos na linha de flags da apresentação do comando R. Estes bits de status nos fornecem valiosas informações sobre a última operação aritmética. Olhando para o flag de zero, por exemplo, podemos dizer se o resultado da última operação foi zero. Também descobrimos que podemos comparar dois números com uma instrução CMP.

A seguir, vimos como imprimir um número hexa com um único dígito. Depois, com essa informação em mãos, passamos para a instrução SHR, que nos permite mover o dígito da direita para os quatro bits da esquerda de um byte. Com isso feito, pudemos imprimir o dígito, assim como tínhamos feito antes.

Finalmente, descobrimos que a instrução AND nos permite isolar um dígito hexa do outro, e, ajuntando tudo isso, escrevemos um programa para imprimir um número com dois dígitos hexa.

Poderíamos ter continuado para imprimir um número hexa de quatro dígitos, mas neste ponto iríamos repetir muitas instruções. Antes de tentarmos imprimir um número hexa de quatro dígitos, veremos as sub-rotinas no Capítulo 7. Depois, saberemos o bastante para escrevermos uma rotina para fazer o serviço. Aí então também estaremos prontos para aprender sobre o assembler — um programa que realizará muito trabalho em nosso lugar. Mas, agora, vamos passar para a leitura de números hexadecimais.

6

LENDO CARACTERES

Lendo um Caractere

Lendo um Número Hexa de Único Dígito

Lendo um Número Hexa de Dois Dígitos

Resumo

Agora que sabemos como imprimir um byte em notação hexa, vamos inverter o processo lendo dois caracteres — dígitos hexa — do teclado, convertendo-os em um único byte.

Lendo um Caractere

A função INT 21h de chamada do DOS que usamos possui uma função de entrada, número 1, que lê um caractere do teclado. Quando aprendemos sobre as chamadas de função no Capítulo 4, vimos que o número da função deve ser colocado no registrador AH antes de uma chamada INT 21h. Vamos tentar a função 1 para INT 21h. Entre com INT 21h na posição 0100h:

```
396F:0100 CD21          INT     21
```

Depois, coloque 01h em AH e digite ou *G 102* ou *P* para executar esta única instrução de chamada. Não acontece nada? Não parece — tudo que você verá é um cursor piscando. Na verdade, porém, o DOS parou e está esperando que você pressione uma tecla (não o faça ainda). Quando você pressionar uma tecla, o DOS colocará o código ASCII desse caractere no registrador AL. Usaremos esta instrução mais tarde, para ler o caractere de um número hexa, mas agora vejamos o que acontece quando você pressiona uma tecla como F1.

Tente pressionar a tecla F1. O DOS retornará um 0 em AL, e você também verá um sinal de ponto-e-vírgula (;) logo após o sinal de pronto do Debug.

Veja o que aconteceu. F1 é uma de um grupo de teclas especiais com *código estendidos*, que o DOS trata de forma diferente das teclas que representam caracteres ASCII normais. (Você verá uma tabela com estes códigos estendidos no Apêndice D, e possivelmente no final do seu manual de BASIC.) Para cada uma destas teclas especiais, o DOS envia dois caracteres, um após o outro. O primeiro caractere retornado é sempre 0, indicando que o próximo caractere é o *código de varredura* de uma tecla em especial.

Para ler ambos os caracteres, precisaríamos executar a instrução INT 21h duas vezes. Mas no nosso exemplo nós lemos somente o primeiro caractere, o zero, e deixamos o código de varredura no DOS. Quando o Debug terminou com o comando G 102 (ou P), ele começou a ler caracteres, e o primeiro caractere que ele leu foi o código de varredura deixado pela tecla F1: a sabe, 59 (3B hexa), que é o código ASCII do ponto-e-vírgula.

Mais tarde, quando desenvolvermos nosso programa Dskpatch, começaremos a usar estes códigos estendidos para ativar o cursor e as teclas de função. Até lá, trabalharemos somente com os caracteres ASCII normais.

Lendo um Número Hexa de Único Dígito

Vamos inverter a conversão usada no Capítulo 5, quando transformamos um número hexa de um dígito para o código ASCII de um dos caracteres de 0 a 9 ou de A a F. Para converter um caractere, como C ou D, de um caractere hexa para um byte, devemos subtrair ou 30h (para dígitos de 0 a 9) ou 37h (de A a F). Eis um programa simples que pode ler um caractere ASCII e convertê-lo para um byte:

```
3985:0100 B401          MOV     AH,01
3985:0102 CD21          INT     21
3985:0104 2C30          SUB     AL,30
3985:0106 3C09          CMP     AL,09
3985:0108 7E02          JLE     010C
3985:010A 2C07          SUB     AL,07
3985:010C CD20          INT     20
```

A maioria das instruções já devem ser conhecidas agora, mas há uma nova, JLE (*Jump if Less Than or Equal to — Salta se for Menor ou Igual*). Em nosso programa, esta instrução salta se AL for menor ou igual a 9h.

Para ver a conversão do caractere hexa para ASCII, você precisa ver o registrador AL antes que INT 20h seja executado. Como o Debug recupera os registradores quando ele executa o INT 20h, você precisará definir um breakpoint, como foi feito no Capítulo 4. Aqui, digite *G 10C*, e você verá que o registrador AL contém o número hexa convertido do caractere.

Tente digitar alguns caracteres, como k ou d minúsculos, que não são dígitos hexa, para ver o que acontece. Você notará que este programa só funciona corretamente quando a entrada for um dos dígitos de 0 a 9 ou as letras maiúsculas de A a F. Corrigiremos esta pequena falha no próximo capítulo, quando falarmos sobre sub-rotinas, ou procedures. Até lá, vamos passar em frente e ignorar condições de erro: temos que digitar os caracteres corretos para que o programa funcione perfeitamente.

Lendo um Número Hexa de Dois Dígitos

A leitura de dois dígitos hexa não é muito mais complicada que a leitura de um, mas requer muito mais instruções. Começaremos lendo o primeiro dígito, depois colocaremos seu valor hexa em DL e o multiplicaremos por 16. Para executar esta multiplicação, iremos deslocar o registrador DL quatro bits à esquerda, colocando um zero hexa (quatro bits zero) à direita do dígito que acabamos de ler. A instrução SHL DL,CL, com CL contendo 4 fará este truque inserindo zeros à direita. Na verdade, a intrução SHL (*Shift Left — Deslocar à Esquerda*) é conhecida como *deslocamento aritmético*, pois ela tem o mesmo efeito de se multiplicar por dois, quatro, oito, dezesseis, e assim por diante, dependendo do número em CL (como um, dois, três etc.).

Figura 6-1. ***A Instrução SHL DL, 1.***

Finalmente, com o primeiro dígito deslocado, podemos adicionar o segundo dígito hexa ao número em DL (que é o primeiro dígito X 16). Você poderá ver e estudar todos estes detalhes com este programa:

```
3985:0100 B401      MOV     AH,01
3985:0102 CD21      INT     21
3985:0104 88C2      MOV     DL,AL
3985:0106 80EA30    SUB     DL,30
3985:0109 80FA09    CMP     DL,09
3985:010C 7E03      JLE     0111
3985:010E 80EA07    SUB     DL,07
3985:0111 B104      MOV     CL,04
3985:0113 D2E2      SHL     DL,CL
3985:0115 CD21      INT     21
3985:0117 2C30      SUB     AL,30
3985:0119 3C09      CMP     AL,09
3985:011B 7E02      JLE     011F
3985:011D 2C07      SUB     AL,07
3985:011F 00C2      ADD     DL,AL
3985:0121 CD20      INT     20
```

Agora já temos um programa para trabalhar, e novamente é uma boa idéia checar as condições de limite para confirmarmos que ele está funcionando corretamente. Para estas condições de limites, use os números 00, 09, 0A, 0F, 90, A0, F0, e algum outro número, como, por exemplo, 3C. Use um ponto de interrupção para executar o programa sem executar a instrução INT 20h. (Lembre-se de usar letras maiúsculas para suas entradas em hexa.)

Resumo

Finalmente tivemos uma chance de praticar o que aprendemos nos capítulos anteriores sem que houvesse uma enchente de novas informações. Usando uma nova função de INT 21 (a de número 1) para ler caracteres do teclado, desenvolvemos um programa para ler um número hexa de dois dígitos. Durante o processo, enfatizamos a necessidade de testarmos os programas com todas as condições de limite.

Agora já estamos prontos para encerrar a Parte I aprendendo sobre as sub-rotinas no 8088.

7

PROCEDURES PRIMAS DAS SUB-ROTINAS

Procedures

A Pilha e os Endereços de Retorno

PUSH e POP

Lendo Números Hexa com Mais Facilidade

Resumo

No próximo capítulo, veremos algo sobre MASM, o macroassembler, e começaremos a usar a linguagem *assembly*, ou *assembler*. Mas antes de deixarmos o Debug, veremos um último grupo de exemplos, e aprenderemos sobre as sub-rotinas e um lugar especial para armazenar números, chamado pilha.

Procedures

Uma procedure e uma lista de instruções que podemos executar de muitos lugares diferentes do programa, ao invés de ter que ser repetida em todos os lugares em que ela seja necessária. Em BASIC, tais listas são conhecidas como *sub-rotinas*, mas chamaremos de *rotinas* por razões que se tornarão evidentes mais tarde.

Entramos e saímos das rotinas assim como no BASIC. Chamamos uma rotina com uma instrução, CALL, que é igual ao GOSUB do BASIC. Retornamos da rotina com uma instrução RET, que é igual ao RETURN do BASIC.

Eis um programa simples em BASIC que brevemente iremos passar para linguagem de máquina. Este programa chama uma sub-rotina dez vezes, cada vez imprimindo um caractere, começando com A e terminando com J:

```
10 A = &H41          ASCII de 'A'
20 FOR I = 1 TO 10
30 GOSUB 1000
40 NEXT I
50 END
1000 PRINT CHR$ (A),
1100 A = A + 1
1200 RETURN
```

A sub-rotina, seguindo uma prática comum nos programas em BASIC, começa na linha 1000 e deixa espaço para que acrescentemos mais instruções ao programa principal sem afetar nossa rotina. Faremos o mesmo com a rotina em linguagem de máquina colocando-a em 200h, longe do nosso programa principal em 100h. Também trocaremos o GOSUB 1000 pela instrução CALL 200h, que *chama* a rotina na posição de memória 200h. O CALL passa o registrador IP para 200h, e o 8088 começa a executar as instruções em 200h.

O loop FOR-NEXT do programa em BASIC, como vimos no Capítulo 4, pode ser escrito com uma instrução LOOP. As outras partes do programa principal devem ser conhecidas.

```
3985:0100 B241        MOV     DL,41
3985:0102 B90A00      MOV     CX,000A
3985:0105 E8F800      CALL    0200
3985:0108 E2FB        LOOP    0105
3985:010A CD20        INT     20
```

A prímeira instrução coloca 41h (ASCII de A) no registrador DL, pois a instrução INT 21h imprime o caractere dado pelo código ASCII em DL. A instrução INT 21h é localizada a certa distância, na rotina localizada em 200h. Eis a rotina que deve entrar em 200h:

```
3985:0200 B402        MOV     AH,02
3985:0202 CD21        INT     21
3985:0204 FEC2        INC     DL
3985:0206 C3          RET
```

Há duas instruções novas e duas antigas aqui. Lembre-se de que 02h em AH diz ao DOS para imprimir o caractere em DL quando executarmos a instrução INT 21h. INC DL, a primeira das duas novas instruções, *incrementa* o registrador DL. Isto é, ela soma 1 a DL. A outra nova instrução, RET, *retorna* para a primeira instrução (LOOP) após o CALL do nosso programa principal.

Digite G para ver a saída do programa, depois verifique passo-a-passo o que ele faz (lembre-se de usar ou um breakpoint ou o comando P para executar a instrução INT 21).

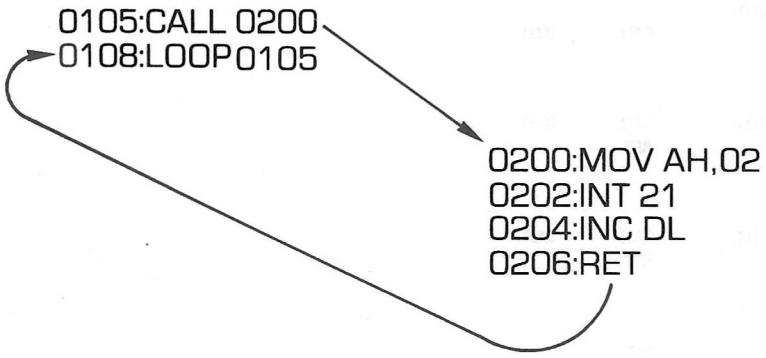


Figura 7-1. ***As Instruções CALL e RET.***

A Pilha e os Endereços de Retorno

A instrução CALL em nosso programa precisa salvar o *endereço de retorno* em algum lugar para que o 8088 saiba onde continuar a execução das instruções quando encontrar a instrução RET. Para o lugar do armazenamento em si, temos uma parte da memória conhecida como pilha. E para verificar o que há na pilha, há dois registradores que podemos ver com o comando R: o registrador SP (*Stack Pointer — Ponteiro de Pilha*), que aponta para o topo da pilha, e o SS (*Stack Segment — Segmento de Pilha*), que contém o número do segmento onde a pilha se encontra.

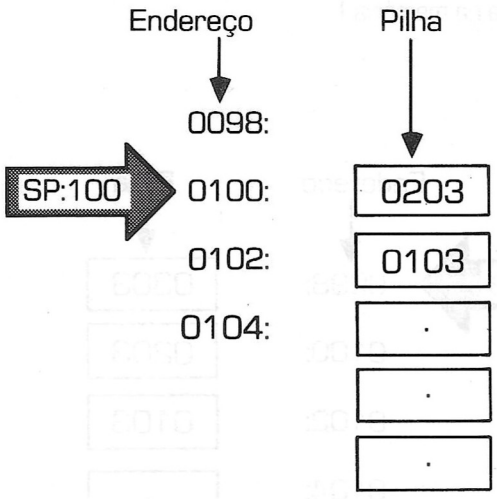


Figura 7-2. ***A Pilha Antes de Executar a Instrução CALL 400.***

Em operação, uma pilha para o 8088 é algo como uma pilha de pratos em um restaurante, onde a colocação de um prato no alto cobre os pratos de baixo. O último prato na pilha é o primeiro a sair, e por isso a pilha é qualificada como LIFO, que significa *Last In, First Out* (*último a entrar, primeiro a sair*). Esta ordem, LIFO, é exatamente o que precisamos para recuperar os endereços de retorno após fazermos chamadas *endentadas* com a instrução CALL:

```
396F:0100 E8FD00
                        CALL    0200
                          .
                          .
                          .
396F:0200 E8FD00        CALL    0300
396F:0203 C3            RET
                          .
                          .
                          .
396F:0300 E8FD00        CALL    0400
396F:0303 C3            RET
                          .
                          .
                          .
396F:0400 C3            RET
```

Aqui, a instrução 100h chama uma outra em 200h, que chama uma em 300h, que chama uma em 400h, onde finalmente vemos uma instrução de retorno (RET). Este RET volta à instrução após a instrução CALL *anterior*, em 300h, de modo que o 8088 termina a execução das instruções em 303h. Mas lá ele encontra uma instrução RET em 303h, que retira o próximo endereço mais antigo (203h) da pilha. Assim, o 8088 continua a executar as instruções em 203h, e assim por diante. Cada RET *retira* (*pop*) o mais alto endereço de retorno da pilha, de modo que cada RET percorre o mesmo caminho das instruções CALL ao contrário, retornando à primeira instrução que usou o CALL.

Tente usar um programa como esse anterior. Use chamadas múltiplas, e acompanhe cada instrução separadamente para ver como as chamadas e retornos funcionam. Embora o processo possa não parecer muito interessante nesse momento, há outras possibilidades para essa pilha, e será muito bom você ter um bom conhecimento de como ela funciona. (Num dos capítulos a seguir, iremos procurar o local da pilha na memória.)

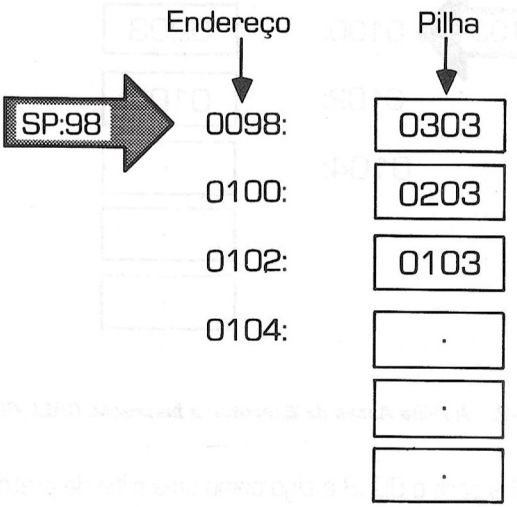


Figura 7-3. ***A Pilha Logo Após a Execução da Instrução CALL 400.***

PUSH e POP

A pilha é um lugar prático para se colocar palavras de dados por um certo tempo, desde que recuperemos esses dados antes de uma instrução RET. Já vimos que uma instrução CALL *coloca (pushes)* o endereço de retorno no topo da pilha, carrega-o no registrador IP, enquanto uma instrução RET *retira (pops)* esta palavra do topo da pilha, coloca-a no registrador IP, e deixa exposta a palavra colocada imediatamente abaixo dela na pilha. Podemos fazer a mesma coisa com as instruções PUSH e POP, que nos permite colocar e retirar palavras da pilha. Mas quando vamos precisar disso?

Sempre é conveniente salvar os valores dos registradores no início de uma rotina e recuperá-los no final, logo antes da instrução RET. Depois estamos livres para usar estes registradores da forma que quisermos dentro da rotina, desde que recuperemos seus valores ao final.

Os programas são montados em diversos níveis de sub-rotinas, com cada nível chamando as rotinas no próximo nível mais baixo. Salvando os registradores no início de uma rotina e recuperando-os ao final, removemos interações indesejadas entre rotinas de níveis diferentes, e isso torna nossa tarefa de programação muito mais fácil. Você verá mais sobre como salvar e recuperar registradores no Capítulo 13, quando falarmos sobre o projeto modular. Por enquanto, veja um exemplo (não entre com ele) de como salvar e recuperar CX e DX:

```
396F:0200 51          PUSH    CX
396F:0201 52          PUSH    DX
396F:0202 B90800      MOV     CX,0008
396F:0205 E8F800      CALL    0300
396F:0208 FEC2        INC     DL
396F:020A E2F9        LOOP    0205
396F:020C 5A          POP     DX
396F:020D 59          POP     CX
396F:020E C3          RET
```

Note que os POPs estão na ordem contrária dos PUSHes, pois um POP remove a palavra colocada por último na pilha, e o valor antigo do DX está acima de CX.

Salvando e recuperando CX e DX, podemos alterar estes registradores à vontade na rotina que começa em 200, mas sem alterar os valores usados por qualquer rotina que chamar esta. E uma vez salvos CX e DX, podemos usar estes registradores para conter variáveis *locais* — variáveis que podemos usar dentro desta rotina sem afetar os valores usados pelo programa que a chamou.

Usaremos tais variáveis locais para simplificar nossas tarefas de programação. Enquanto tivermos o cuidado de restaurar os valores originais, não precisamos nos preocupar com a alteração dos registradores usados pelo programa que chama uma rotina qualquer. Isto ficará mais claro no exemplo seguinte, que é uma rotina para ler um número hexa. Diferente do programa no Capítulo 6, nosso programa agora permitirá somente caracteres válidos, como A, mas não K.

Lendo Números Hexa com Mais Facilidade

Queremos criar uma rotina que fique lendo caracteres até que receba um que possa ser convertido para um número hexa entre 0 e Fh. Não queremos apresentar caracteres inválidos, e por isso iremos modificar nossa entrada usando uma função nova, a de número 8, que lê um caractere mas não o passa para a tela. Dessa forma, podemos *ecoar* (apresentar) os caracteres somente se eles forem válidos.

Coloque 8h no registrador AH e execute esta instrução, digitando um A logo após digitar G 102:

```
3985:0100 CD21        INT     21
```

O código ASCII de A (41h) está agora no registrador AL, mas o A não aparece na tela.

Usando esta função, nosso programa pode ler caracteres sem colocá-los na tela até que verifique se o caractere é um dígito hexa válido (de 0 a 9 ou de A a F), quando então o apresentará. Eis a rotina para fazer isso e converter o caractere hexa em um número hexa:

```
3985:0200 52          PUSH    DX
3985:0201 B408        MOV     AH,08
3985:0203 CD21        INT     21
3985:0205 3C30        CMP     AL,30
3985:0207 72FA        JB      0203
3985:0209 3C46        CMP     AL,46
3985:020B 77F6        JA      0203
3985:020D 3C39        CMP     AL,39
3985:020F 770A        JA      021B
3985:0211 B402        MOV     AH,02
3985:0213 8BC2        MOV     DL,AL
3985:0215 CD21        INT     21
3985:0217 2C30        SUB     AL,30
3985:0219 5A          POP     DX
3985:021A C3          RET
3985:021B 3C41        CMP     AL,41
3985:021D 72E4        JB      0203
3985:021F B402        MOV     AH,02
3985:0221 8BC2        MOV     DL,AL
3985:0223 CD21        INT     21
3985:0225 2C37        SUB     AL,37
3985:0227 5A          POP     DX
3985:0228 C3          RET
```

A rotina lê um caractere em AL (com o INT 21h em 203h) e verifica se ele é válido com os CMPs e os saltos condicionais. Se o caractere lido não for válido, as instruções de salto condicional enviam o 8088 de volta à posição 203, onde o INT 21h lê outro caractere. (JA significa *Jump if Above — salto se Acima;* JB significa *Jump if Below — salto se Abaixo;* ambos tratam os dois números como sendo números sem sinal, enquanto a instrução JL que usamos anteriormente tratava ambos como números com sinal.)

Na linha 211h, já sabemos que há um dígito válido entre 0 e 9, de modo que subtraímos do código do caractere 0 e retornamos o resultado no registrador AL, lembrando de recuperar o registrador DX, que foi salvo no início da rotina. O processo para os dígitos hexa de A a F é muito parecido. Note que temos duas instruções RET nesta rotina; poderíamos ter mais, ou então poderíamos ter somente uma.

Eis um programa muito simples para testar a rotina:

```
3985:0100 E8FD00      CALL    0200
3985:0103 CD20        INT     20
```

Asssim como você já fez antes, use ou o comando G com um breakpoint, ou então o comando P. Você vai querer executar a instrução CALL 200h sem executar a instrução INT 20h; assim você pode ver os registradores antes do término do programa e da recuperação dos registradores.

Você verá o cursor no lado esquerdo da tela, esperando pacientemente por um caractere. Digite *k*, que não é um caractere válido. Nada deverá acontecer. Agora, digite um dos caracteres hexa maiúsculos. Você deverá ter o valor hexa do caractere em AL e o próprio caractere na tela. Teste esta rotina com as condições de limite: " \" (o caractere antes de 0), 0, 9, ":" (o caractere logo após 9), e assim por diante.

Agora que temos esta rotina, o programa para ler um número hexa de dois dígitos, com manipulação de erro, é bem simples:

```
3985:0100 E8FD00        CALL    0200
3985:0103 88C2          MOV     DL,AL
3985:0105 B104          MOV     CL,04
3985:0107 D2E2          SHL     DL,CL
3985:0109 E8F400        CALL    0200
3985:010C 00C2          ADD     DL,AL
3985:010E B402          MOV     AH,02
3985:0110 CD21          INT     21
3985:0112 CD20          INT     20
```

Você pode executar este programa pelo DOS, pois ele lê um número hexa de dois dígitos e depois apresenta o caractere ASCII que corresponde ao número que você digitou.

Sem contar a rotina, nosso programa é mais simples do que a versão que escrevemos no capítulo anterior, e não duplicamos as instruções para ler os caracteres. Pudemos criticar os erros, no entanto, e mesmo que isso tenha complicado nossa rotina, também assegurou que o programa agora aceita somente entradas válidas.

Aqui também podemos ver a razão para salvar o registrador DX na rotina. O programa principal armazena o número hexa em DL, e não queremos que nossa rotina em 200h altere DL. Por outro lado, a rotina em 200h deve usar o próprio DL para apresentar os caracteres. Assim, usando a instrução PUSH DX no início da rotina, e POP DX no final, podemos nos livrar de qualquer problema que apareça.

De agora em diante, para evitar interações complicadas entre rotinas, seremos bem exigentes quanto à questão de salvar os registradores usados por uma rotina.

Resumo

Nossa programação está se tornando mais sofisticada. Já aprendemos sobre as rotinas que nos permitem reutilizar o mesmo grupo de instruções sem ter que reescrevê-las a cada vez. Também descobrimos a pilha e vimos que um CALL armazena um endereço de retorno no topo da pilha enquanto uma instrução RET retorna ao endereço contido no topo da pilha.

Vimos como usar a pilha para outras tarefas além de salvar endereços de retorno. Usamos a pilha para armazenar os valores dos registradores (com uma instrução PUSH) para que pudéssemos usá-los na rotina, evitamos interações indesejadas entre as rotinas. Sempre salvando e recuperando os registradores em nossas rotinas, podemos chamar outras rotinas sem nos preocuparmos com os registradores que são usados nessa outra rotina.

E finalmente, com tudo isso em mente, criamos um programa melhor para ler dígitos hexa — desta vez, com checagem de erro. O programa que fizemos é semelhante a um que usaremos em capítulos posteriores, quando começaremos a desenvolver o programa Dskpatch.

Agora estamos prontos para passar à Parte II, onde aprenderemos como usar o assembler. No capítulo a seguir, veremos como usar o assembler para converter um programa para linguagem de máquina. Também veremos que não há qualquer razão para deixarmos espaço entre as rotinas, como fizemos neste capítulo, quando colocamos nossa rotina somente na posição 200h.

PARTE II

LINGUAGEM ASSEMBLY

8

BEM-VINDO AO ASSEMBLER

Um Programa Sem Debug

Criando Arquivos Fonte

Linkeditando

De Volta ao Debug

Comentários

Rótulos

Resumo

Bem, depois de tudo o que vimos, estamos finalmente prontos para ver o assembler, um programa do DOS que tornará nossa programação muito mais simples. Daqui em diante, escreveremos com mnemônicos, instruções diretamente legíveis, usando o assembler para passar nossos programas para o código de máquina.

Por necessidade, este capítulo e o próximo terão muitos detalhes sobre o assembler, mas os esforços para este aprendizado valerá a pena depois. Uma vez sabendo como usar o assembler, voltaremos a aprender como escrever programas em linguagem assembly. Nesse meio tempo, vamos nos aprofundar nos estudos.

Um Programa Sem Debug

Até aqui, sempre digitamos *DEBUG*, e depois nossas instruções de programa. Agora iremos deixar o Debug para trás, e escreveremos programas sem ele, usando ou um editor ou um processador de texto para criar os arquivos de texto contendo nossas instruções em linguagem assembly.

Começamos criando um *arquivo fonte* — o nome para a versão texto de um programa em linguagem assembly. Criaremos agora um arquivo fonte, para o programa que montamos no Capítulo 3, e o chamaremos Impstr. Para refrescar sua memória, veja nossa versão do Debug:

```
396F:0100 B402      MOV     AH,02
396F:0102 B261      MOV     DL,2A
396F:0104 CD21      INT     21
396F:0106 CD20      INT     20
```

Use o seu editor para entrar com as seguintes linhas de código em um arquivo chamado IMPSTR.ASM (a extensão .ASM significa que este é um arquivo fonte do assembler). Aqui, assim como no Debug, as minúsculas trabalham assim como as maiúsculas, mas continuaremos a usar maiúsculas para evitar confusão entre o 1 (um) e a letra minúscula l (ele):

```
CODE_SEG        SEGMENT
        MOV     AH,2h
        MOV     DL,2Ah
        INT     21h
        INT     20h
CODE_SEG        ENDS
        END
```

Este é o mesmo programa que criamos no Capítulo 3, mas possui algumas mudanças e acréscimos necessários. Ignore por enquanto, as três linhas novas desse arquivo fonte, e note que há um *h* após cada número hexa. Este *h* diz ao assembler que os números estão em hexadecimal. Diferente do Debug, que assume que todos os números estão em hexadecimal, o assembler assume que todos os números estão em decimal. Isso é modificado colocando-se um "h" após qualquer número em notação hexadecimal.

Nota: Antes de prosseguirmos, note o seguinte: O Assembler pode confundir-se com os números, como ACh, que se parece com um nome ou uma instrução. Para evitar isso, sempre digite um zero antes de qualquer número que comece com uma letra. Por exemplo, digite 0ACh — e *não* ACh.

Veja o que acontece quando montamos um programa com ACh, ao invés de 0ACh:

```
CODE_SEG        SEGMENT
        MOV     DL,ACh
        INT
                20h
CODE_SEG        ENDS
        END
```

Veja agora a saída:

```
A>MASM TEST;
Microsoft (R) Macro Assembler Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985. All rights reserved.

TEST.ASM(2): error 9: Symbol not defined AC

  51070 Bytes symbol space free

      0 Warning Errors
      1 Severe Errors

A>
```

Definitivamente não é nada encorajador. Mas mude ACh por 0ACh, e o assembler ficará muito satisfeito.

Note também o espaçamento dos comandos em nosso programa assembler. Usamos tabs para alinhar tudo e tornar o texto fonte bem mais legível. Compare o programa em que você entrou com esta versão:

```
CODE_SEGMENT
MOV AH,2h
MOV DL,2Ah
INT 21h
INT 20h
CODE_SEG ENDS
END
```

Um tanto confusa; o assembler não se importa, mas nós sim.

Figura 8-1. ***Coloque um zero antes dos números hexadecimais que começam com uma letra, ou o assembler tratará esse número como se fosse um nome.***

Agora vamos ver as três linhas novas no código fonte. As três linhas são todas *pseudo-operações*. Elas são simplesmente chamadas assim porque, ao invés de gerarem instruções, elas simplesmente fornecem informação ao assembler que está traduzindo o código fonte para linguagem de máquina. A pseudo-op END marca o fim do código fonte, para que o assembler possa saber quando termina o código fonte. Mais adiante, veremos que o END também é útil para outras coisas. Mas por enquanto, vamos deixar de lado qualquer discussão a mais sobre isso ou sobre as outras pseudo-ops para vermos como usar o assembler.

Criando Arquivos Fonte

Embora você já tenha entrado com as linhas do programa IMPSTR.ASM, há mais uma coisa que devemos considerar antes de continuarmos para realmente traduzir nosso programa. O assembler pode usar arquivos fonte somente se tiverem caracteres ASCII-padrão. Se você está usando um processador de texto, tenha em mente que nem todos os processadores de texto gravam arquivos em disco usando somente caracteres ASCII-padrão. O WordStar é um desses criminosos; o Word da Microsoft é outro. Para esses dois processadores, use o modo "non-document", ou não-formatado, quando salvar seus arquivos.

Antes de tentar montar IMPSTR.ASM, assegure-se de que ele ainda está em ASCII. A partir do DOS, digite:

```
A>TYPE IMPSTR.ASM
```

Você deverá ver o mesmo texto que entrou, conforme você entrou. Se você encontrar caracteres estranhos no programa, você pode ter que usar um editor ou processador de texto diferente para entrar com os programas. Você também precisará de uma linha em branco após o comando END em seu arquivo.

Agora, vamos começar a montar Impstr (não se esqueça de incluir o ponto-e-vírgula).

```
A>MASM. IMPSTR;
The IBM Personal Computer Assembler
Version 1.00 (C) Copyright IBM Corp 1981

Warning Severe
Errors  Errors
0       0

A>
```

Mas ainda não terminamos. Neste ponto, o assembler criou o arquivo chamado IMPSTR.OBJ, que você pode encontrar no seu disco. Este é um arquivo intermediário, chamado *arquivo objeto*. Ele contém nosso programa em linguagem de máquina, juntamente com uma série de informações usadas por outro programa do DOS chamado *Linkeditor*.

Linkeditando

Agora iremos precisar do linkeditor para pegar nosso arquivo .OBJ e criar sua versão .EXE. Copia o LINK.EXE do disco do DOS para o disco que contém o seu arquivo fonte e o assembler. Depois, linkedite IMPSTR.OBJ digitando:

```
A>LINK IMPSTR;

IBM Personal Computer Linker
Version 1.10 (C) Copyright IBM Corp 1982

Warning: No STACK segment

There was 1 error detected

A>
```

Um erro? Não exatamente; o linkeditor conta o seu aviso (warning) como um erro, mas nesse caso é realmente isso que queremos. (Em algumas versões do MS-DOS, o linkeditor não informa este aviso como um erro.) Mesmo o linkeditor nos avisando que não há um segmento de pilha, não precisaremos dele agora. Após termos aprendido mais alguns truques, veremos quando será preciso um segmento de pilha.

Agora já temos nosso arquivo .EXE, mas esse ainda não é o último passo. Há mais um passo — criar uma versão .COM, que é exatamente o que criamos com o Debug. Novamente, você verá mais tarde por que precisamos de todos esses passos. Por enquanto, vamos criar uma versão .COM de Impstr.

Para o nosso passo final, precisamos do programa EXE2BIN.EXE do disco complementar do DOS. EXE2BIN, como o nome sugere, converte um arquivo .EXE para um arquivo .COM, ou arquivo binário (bin). Há uma diferença entre arquivos .EXE e .COM, mas veremos isso mais tarde, e por ora vamos simplesmente criar o arquivo .COM. Digite:

```
A>EXE2BIN IMPSTR IMPSTR.COM

A>
```

A resposta não é muito esclarecedora. Para ver se Exe2bin funcionou, vamos listar todos os arquivos Impstr que criamos até aqui:

```
A>DIR IMPSTR.*

 Volume in drive A has no label
 Directory of A:\

IMPSTR ASM       78      11-25-87   5:00p
IMPSTR OBJ       46      11-25-87   7:02p
IMPSTR EXE      640      11-25-87   7:04p
IMPSTR COM        8      11-25-87   7:06p
       7 file(s)       23552 bytes free

A>
```

Como apareceu arquivo! E inclui o IMPSTR.COM. Digite *impstr* para executar a versão .COM e verificar se o programa funciona corretamente (lembre-se que ele deve imprimir um asterisco na tela). Os tamanhos exatos que o DOS informa para os primeiros três arquivos podem variar um pouco.

Os resultados podem parecer poucos até aqui, visto que voltamos ao que havia no Capítulo 3, mas não: ganhamos muita coisa. Tudo ficará muito mais claro quando passarmos a usar as chamadas novamente. Note que não precisamos mais nos preocupar com o local onde o programa está localizado na memória, como fizemos alterando IP no Debug. Os endereços foram todos completados para nós.

Logo você irá apreciar esta característica no assembler: ela tornará a programação muito mais fácil. Por exemplo, lembre-se que no capítulo anterior nós perdemos espaço deixando o programa no endereço 100h e a rotina em 200h. Veremos que o assembler nos permite colocar uma rotina imediatamente após um programa principal sem qualquer intervalo entre eles. Mas primeiro, vejamos como o nosso programa se apresenta aos "olhos" do Debug.

De Volta ao Debug

Vamos ler nosso arquivo .COM no Debug e desassemblá-lo para ver como o Debug reconstrói nosso programa a partir do código de máquina de IMPSTR.COM:

```
A>DEBUG IMPSTR.COM
-U
397F:0100 B402      MOV     AH,02
397F:0102 B22A      MOV     DL,2A
397F:0104 CD21      INT     21
397F:0106 CD20      INT     20
```

Exatamente o que tínhamos no Capítulo 3. Isto é tudo que o Debug vê em IMPSTR.COM. O END e nossas instruções adicionais sobre segmentos — CODE_SEGMENT e CODE_SEG ENDS — não aparecem de forma alguma. O que aconteceu a elas?

Estas instruções não aparecem na versão final do programa em linguagem de máquina porque elas são pseudo-ops, e as pseudo-ops servem somente para o uso do assembler. O assembler toma conta de muita coisa às custas de algumas linhas a mais. Iremos utilizar bastante as pseudo-ops para simplificar nosso trabalho, e veremos como elas afetam nosso programa, quando virmos mais de perto os segmentos no Capítulo 11.

Comentários

Como não estamos mais operando diretamente com o Debug, somos livres para acrescentar mais coisas aos nossos programas, que o assembler vê mas não passa adiante para o 8088. Talvez a coisa mais importante que podemos adicionar são os comentários, que são de valor imenso para a clareza dos programas. Em programas em linguagem assembly, colocamos comentários após um sinal de ponto-e-vírgula, que funciona como um apóstrofo (') no BASIC. O assembler ignora tudo na linha após o ponto-e-vírgula, de modo que podemos acrescentar qualquer coisa que desejarmos. Se colocarmos comentários em nosso pequeno programa:

```
CODE_SEG        SEGMENT
        MOV     AH,2h       ;Seleciona funcao 2 do DOS, saida caractere
        MOV     DL,2Ah      ;Carrega o cod. ASCII para "*" ser impresso
        INT
                21h         ;Imprime o asterisco com INT 21h
        INT
                20h         ;... e sai para o DOS.
CODE_SEG
        END
```

podemos ver uma grande melhoria — podemos entender o programa sem ter que pensar muito e lembrar o que significa cada linha.

Rótulo

Para fechar este capítulo, vejamos outra característica do assembler especificamente que torna nosso trabalho mais simples: os rótulos.

Até aqui, quando queríamos saltar de uma parte do programa para outra com um dos comandos de salto, tivemos que saber o endereço específico para onde íamos saltar. Quando estamos programando, a inclusão de novas linhas nos força a alterar os endereços nas instruções de salto. O assembler cuida desse problema com *rótulos* ("labels") — nomes que damos ao endereço de qualquer instrução ou posição de memória. Um rótulo toma o lugar de um endereço. Quando o assembler encontra um rótulo, ele troca o rótulo pelo endereço correto antes de enviá-lo para o 8088.

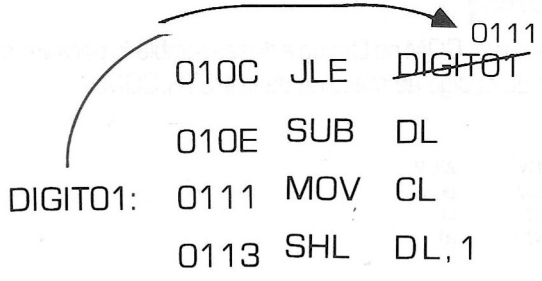


Figura 8-2. ***O Assembler Substitui os Rótulos Pelos Endereços.***

Os rótulos podem ter até 31 caracteres e podem conter letras, números, e qualquer um dos seguintes símbolos: ponto de interrogação (?), ponto (.), arroba (@), um caractere de sublinhar (_) ou um sinal de cifrão. Eles não podem iniciar com um dígito (de 0 a 9), e um ponto só pode ser usado se for o primeiro caractere desse rótulo.

Como um exemplo prático, vejamos o nosso programa do Capítulo 6 que lê um número hexa de dois dígitos. Ele contém dois saltos, JLE 0111 e JLE 011F. Eis a versão antiga:

```
3985:0100 B401      MOV     AH,01
3985:0102 CD21      INT     21
3985:0104 88C2      MOV     DL,AL
3985:0106 80EA30    SUB     DL,30
3985:0109 80FA09    CMP     DL,09
3985:010C 7E03      JLE     0111
3985:010E 80EA07    SUB     DL,07
3985:0111 B104      MOV     CL,04
3985:0113 D2E2      SHL     DL,CL
3985:0115 CD21      INT     21
3985:0117 2C30      SUB     AL,30
3985:0119 3C09      CMP     AL,09
3985:011B 7E02      JLE     011F
3985:011D 2C07      SUB     AL,07
3985:011F 00C2      ADD     DL,AL
3985:0121 CD20      INT     20
```

Não está muito claro o que esse programa faz, e se não estiver fresco em sua memória, você pode ter que trabalhar um pouco para entender o programa novamente. Vamos acrescentar rótulos e comentários para vermos mais facilmente a sua função:

```
CODE_SEG        SEGMENT
        ASSUME  CS:CODE_SEG     ; (Explicado no Capítulo 11)
        MOV     AH,1h           ;Seleciona funcao 1 do DOS, entrada
        INT     21h             ;Le um caractere, e retorna cod em AL
        MOV     DL,AL           ;Move codigo ASCII para DL
        SUB     DL,30h          ;Subtrai de 30h para converter
                                ;... digito para 0 - 9
        CMP     DL,9h           ;Foi um digito entre 0 e 9?
        JLE     DIGITO1         ;Sim, temos nosso primeiro digito (quatro bits)
        SUB     DL,7h           ;Nao, subtraimos 7 para converter
                                ;... letra de A - F
DIGITO1:
        MOV     CL,4h           ;Prepara para multiplicar por 16
        SHL     DL,CL           ;Multiplica deslocando
        INT     21h             ;Apanha proximo caractere
        SUB     AL,30h          ;Repete conversao
        CMP     AL,9h           ;Foi um digito entre 0 e 9?
        JLE     DIGITO2         ;Sim, temos nosso segundo digito
        SUB     AL,7h           ;Nao, subtraimos 7 para converter
DIGITO2:
        ADD     DL,AL           ;ADD com segundo digito
        INT     20h             ;... e sai.
CODE_SEG
        MOV     ENDS
        END
```

Os rótulos aqui, DIGITO1 e DIGITO2, são do tipo do tipo conhecido como *NEAR*, pois um sinal de dois-pontos (:) aparece após os rótulos quando eles são definidos. O termo *NEAR* (*próximo*) tem a ver com segmentos, que falaremos no Capítulo 11, junto com os pseudo-ops SEGMENT, ENDS e ASSUME. Aqui, se você montasse o programa acima, e depois o desassemblasse com o comando U do Debug, você veria DIGITO1 trocado por 0111h e DIGITO2 trocado por 011Fh.

Resumo

Esse foi realmente um capítulo interessante. Foi como se estivéssemos entrando em um mundo novo, e, de certa forma, entramos realmente. O assembler é muito mais fácil de se trabalhar do que o Debug, e agora já podemos escrever programas reais, pois o assembler faz grande parte do trabalho adicional para nós (como, por exemplo, cuidar dos endereços e dos segmentos de código etc.)

O que aprendemos neste capítulo? Começamos aprendendo como criar um arquivo fonte e depois passamos aos passos da montagem, linkedição (conversão de arquivo .OBJ para .EXE), e depois a passagem para arquivo .COM, usando um programa simples do Capítulo 3. O programa em linguagem assembly que criamos possuía algumas pseudo-ops, que nunca vimos antes, mas elas se tornarão familiares, quando estivermos mais acostumados a usar o assembler. Na verdade, iremos colocar SEGMENT, ENDS e END em todos os nossos programas em assembly daqui para frente, pois precisamos deles, mesmo que não saibamos a razão antes do Capítulo 11.

A seguir, aprendemos sobre os comentários. Você deve ter pensado em como podemos programar sem eles. Daqui para frente, iremos utilizá-los bastante, pois acrescentam muita legibilidade ao programa, e não iremos economizar.

Finalmente apareceram os rótulos para tornarem os programas ainda mais legíveis. Usaremos todas estas idéias e métodos no restante deste livro. Vamos prosseguir no próximo capítulo vendo como o assembler torna as rotinas mais fáceis de se usar.

9

PROCEDURES E O ASSEMBLER

As Procedures do Assembler

As Procedures de Saída em Hexa

Rudimentos do Projeto Modular

Um Esqueleto de Programa

Resumo

Agora que já conhecemos o assembler, vamos nos acostumar a escrever programas em linguagem assembly. Neste capítulo, voltaremos ao assunto das procedures. Você verá como podemos escrever procedures muito mais facilmente com nosso amigo assembler. Depois, passaremos a escrever algumas rotinas práticas, que iremos utilizar quando começarmos a desenvolver nosso programa Dskpatch alguns capítulos mais adiante.

Começaremos com duas rotinas para imprimir um byte em hexa. No caminho, iremos encontrar diversas pseudo-ops. Mas, assim como SEGMENT, END e ENDS no capítulo anterior, iremos deixá-las um pouco em aberto até o Capítulo 11, quando aprendermos mais sobre segmentos.

As Procedures do Assembler

Quando aprendemos inicialmente sobre as procedures, demos um grande salto entre o programa principal e suas procedures, para que tivéssemos espaço para mudanças sem que nos preocupássemos com a superposição das procedures pelo programa principal. Mas agora nós temos o assembler, e como ele faz todo o trabalho de atribuir endereços às instruções, não temos mais que deixar um intervalo entre as procedures. Com o assembler, toda vez que fizermos uma mudança, podemos simplesmente montar o programa novamente.

No Capítulo 7, montamos um pequeno programa com um CALL. O programa não fez nada mais do que imprimir as letras de A a J, e ele era mais ou menos assim:

```
3985:0100 B241          MOV     DL,41
3985:0102 B90A00        MOV     CX,000A
3985:0105 E8F800        CALL    0200
3985:0108 E2FB          LOOP    0105
3985:010A CD20          INT     20

3985:0200 B402          MOV     AH,02
3985:0202 CD21          INT     21
3985:0204 FEC2          INC     DL
3985:0206 C3            RET
```

Vamos passar isso para um programa em assembler. Será difícil lermos sem rótulos e comentários, e por isso iremos colocar esses artifícios para tornar nosso programa mais legível:

Listagem 9-1. O Programa IMPRAJ.ASM

```
CODE_SEG        SEGMENT
        ASSUME  CS:CODE_SEG
        ORG     100h                    ;Para arquivo .COM
                                        ;(explicado depois)
IMPR_A_J        PROC NEAR
        MOV     DL,'A'                  ;Comeca com caractere A
        MOV     CX,10                   ;e imprime 10 caracteres
IMPR_LOOP:
        CALL    IMPR_CAR                ;Imprime caractere e passa
                                        ;... para o proximo
        LOOP    IMPR_LOOP               ;Continua para 10 caracteres
        INT     20h                     ;Retorna ao DOS
IMPR_A_J        ENDP

IMPR_CAR        PROC NEAR
        MOV     AH,2                    ;Seta o codigo da funcao
                                        ; para saida de caractere
        INT     21h                     ;Imprime o caractere em DL
        INC     DL                      ;Move para o prox caractere
        RET                             ;Retorna desta rotina
IMPR_CAR        ENDP

CODE_SEG
        ENDS
        END     IMPR_A_J
```

Há quatro novas pseudo-ops aqui: ASSUME, ORG, PROC e ENDP. ASSUME está relacionada aos segmentos, e ORG está relacionada com o modo com que o DOS carrega os programas; veremos mais sobre isso no Capítulo 11.

PROC e ENDP são pseudo-ops para definirem procedures. Como você pode ver, tanto o programa principal quanto a rotina em 200h estão contornados pelas pseudo-ops PROC e ENDP, que, por sua vez, estão contidas entre as pseudo-ops SEGMENT e ENDS (End Segment — fim de segmento).

PROC define o início de uma procedure; ENDP define o seu fim. Assim, no programa principal, IMPR_A_J, podemos trocar nossa instrução CALL 200 com o comando mais legível CALL IMPR_CAR. Basta inserir o nome da procedure, e o assembler atribui os endereços.

As pseudo-ops NEAR e FAR (veja sobre FAR mais adiante) geram informação ao assembler sobre nosso uso dos segmentos. O assembler usa esta informação sempre que estiver em uma instrução CALL, pois há dois tipos de instruções CALL e RET: próximas e distantes ("near" e "far"). Um CALL distante, que não usamos aqui, chama uma rotina contida em outro segmento. Um CALL próximo, por outro lado, chama uma rotina contida no mesmo segmento.

Neste livro, vamos lidar com programas que são contidos em um único segmento de 64K, de modo que todas as nossas rotinas serão NEAR. NEAR informa ao assembler que a rotina está no mesmo segmento de qualquer rotina que a chamar. Quando o assembler vir CALL IMPR_CAR, ele saberá, pelo NEAR em IMPR_CAR PROC NEAR, que IMPR_CAR está no mesmo segmento de IMPR_A_J.

O assembler precisa dessa informação de segmento porque há duas versões de CALL e RET — uma para quando não passamos dos limites de um segmento, e uma para quando passamos. Aqui é óbvio que nossas duas rotinas estão no mesmo segmento, pois colocamos ambas as rotinas entre um par de pseudo-ops de definição de segmento: SEGMENT e ENDS. Mais tarde, quando dividirmos nosso programa em partes que serão colocadas em diversos arquivos fonte diferentes, o uso de NEAR e FAR tornar-se-á mais importante.

Finalmente, como temos duas rotinas, precisamos dizer ao assembler qual dela usar como rotina principal — onde o 8088 deve começar a executar o nosso programa. A pseudo-op END cuida desse detalhe. Escrevendo END IMPR_A_J, dizemos ao assembler que PRINT_A_J é a rotina principal. Mais adiante, veremos que a rotina principal pode estar em qualquer lugar. Por enquanto, no entanto, estaremos lidando com arquivos .COM, e precisamos colocar a rotina principal em primeiro lugar no código fonte do programa.

Pronto, agora você pode continuar, se ainda não o fez, entrando com o programa em um arquivo chamado IMPRAJ.ASM e gerando a versão .COM, usando os mesmos passos usados no capítulo anterior:

```
MASM IMPRAJ;
LINK  IMPRAJ;
EXE2BIN IMPRAJ IMPRAJ.COM
```

Depois tente executar Impraj. (Lembre-se de executar Exe2bin *antes* de executar o programa. Caso contrário, você acabará executando a versão .EXE, que sem dúvida não produzirá os resultados esperados.)

Quando estiver satisfeito, use o Debug para desassemblar o programa e ver como o assembler ajunta as duas rotinas. Lembre-se que podemos ler um arquivo em particular para o Debug digitando seu nome como parte da linha de comando. Por exemplo, podemos digitar *DEBUG IMPRAJ.COM*, e quanto o fizermos, veremos:

```
3985:0100 B241          MOV     DL,41
3985:0102 B90A00        MOV     CX,000A
3985:0105 E80400        CALL    010C
3985:0108 E2FB          LOOP    0105
3985:010A CD20          INT     20
3985:010C 8402          MOV     AH,02
3985:010E CD21          INT     21
3985:0110 FEC2          INC     DL
3985:0112 C3            RET
```

Nosso programa está limpo e ajustado, sem qualquer intervalo entre as duas rotinas.

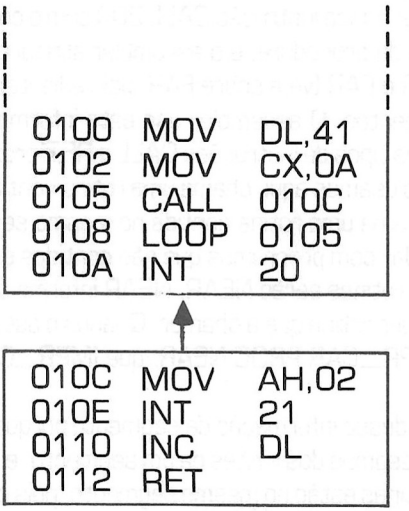


Figura 9-1. ***O MASM Monta Rotinas Separadas Sem Intervalo.***

As Procedures de Saída em Hexa

Já vimos as rotinas de saída em hexa duas vezes antes: uma no Capítulo 5, onde aprendemos como imprimir um número em hexa, e novamente no Capítulo 7, onde vimos como simplificar o programa, usando a rotina para imprimir um dígito hexa. Agora vamos acrescentar ainda outra rotina para imprimir um caractere. Por quê? Bem, vamos tomar uma precaução.

Usando uma rotina central para imprimir um caractere na tela, podemos alterar o modo com que esta rotina imprime os caracteres sem afetar o restante do programa. Iremos mudá-la diversas vezes.

Entre com o seguinte programa no arquivo ES_VIDEO.ASM:

Listagem 9.2. O Novo Arquivo ES_VIDEO.ASM

```
CODE_SEG            SEGMENT
    ASSUME          CS:CODE_SEG
    ORG             100h

TESTE_IMP_HEXA      PROC    NEAR
    MOV             DL,3Fh          Testa com 3Fh
    CALL            IMP_HEXA
    INT             20h             Retorna ao DOS
TESTE_IMP_HEXA      ENDP

    PUBLIC          IMP_HEXA
```

```
;--------------------------------------------------------------------
; Esta rotina converte o byte em DL para hexa e imprime os
; dois digitos hexa na posicao corrente do cursor.
;
;       DL      Byte a ser convertido para hexa
; Usa:          IMP_DIG_HEXA
;--------------------------------------------------------------------
IMP_HEXA        PROC    NEAR    ;Ponto de entrada
        PUSH    CX              ;Salva registradores usados
        PUSH    DX              ;  nesta rotina
        MOV     DH,DL           ;Copia o byte
        MOV     CX,4            ;Coloca nibble alto em DL
        SHR     DL,CL
        CALL    IMP_DIG_HEXA    ;Imprime primeiro digito
        MOV     DL,DH           ;Nibble baixo para DL
        AND     DL,0Fh          ;Remove nibble alto
        CALL    IMP_DIG_HEXA    ;Imprime segundo digito
        POP     DX
        POP     CX
        RET
IMP_HEXA        ENDP

        PUBLIC  IMP_DIG_HEXA


;--------------------------------------------------------------------
; Esta rotina converte os 4 bits baixos de DL para um digito
; hexa e o imprime na tela.
;
;       DL      4 bits baixos contem numero a ser impresso
; Usa:          IMP_CAR
;--------------------------------------------------------------------
IMP_DIG_HEXA    PROC    NEAR
        PUSH    DX              ;Salva registradores usados
        CMP     DL,10           ;O nubble >10?
        JAE     LETRA_HEXA      ;Nao, converte para letra
        ADD     DL,"0"          ;Sim, converte para digito
        JMP     Short IMP_DIGITO ;Imprime este caractere
LETRA_HEXA:
        ADD     DL,"A"-10       ;Converte para letra hexa
IMP_DIGITO:
        CALL    IMP_CAR         ;Imprime letra na tela
        POP     DX              ;Restaura valor de DX
        RET
IMP_DIG_HEXA    ENDP

        PUBLIC  IMP_CAR


;--------------------------------------------------------------------
; Esta rotina imprime um caractere na tela usando a funcao
; de chamada do DOS.
;
;       DL      Byte a imprimir na tela
;--------------------------------------------------------------------
IMP_CAR         PROC    NEAR
        PUSH    AX
        MOV     AH,2            ;Chamada para saida de caractere
        INT     21h             ;Sai caractere no registrador DL
        POP     AX              ;Recupera antigo valor de AX
        RET                     ;E retorna.
IMP_CAR         ENDP

CODE_SEG        ENDS

        END     TESTE_IMP_HEXA
```

A função do DOS para imprimir caracteres trata certos caracteres de forma especial. Por exemplo, usar a função do DOS para sair com 07 resulta em um beep, sem imprimir um caractere para 07. Veremos uma nova versão de IMP_CAR na Parte III, quando aprenderemos sobre as roti-

nas do ROM BIOS dentro do seu IBM PC. Por enquanto, no entanto, só usaremos a função do DOS para imprimir caracteres.

A nova pseudo-op PUBLIC está aqui para uso futuro: iremos usá-la no Capítulo 13, quando aprenderemos sobre o projeto modular. PUBLIC simplesmente diz ao assembler para gerar mais alguma informação para o linkeditor. O linkeditor nos permite juntar, para dentro de um programa, partes separadas do nosso programa, mesmo montadas em diferentes arquivos fonte. E PUBLIC justamente informa ao assembler que a rotina com nome após o PUBLIC deve tornar-se pública, ou seja, disponível às rotinas de outros arquivos.

Agora, Es_video contém as três rotinas para imprimir um byte com número hexa, e um pequeno programa principal para testar estas rotinas. Iremos acrescentar muitas rotinas ao arquivo quando desenvolvermos Dskpatch e, no final deste livro, ES_VIDEO estará repleto com muitas rotinas de uso geral.

A rotina TESTE_IMP_HEXA faz exatamente o que ela diz: ela serve para testar IMP_HEXA, que, por sua vez, usa IMP_DIG_HEXA e IMP_CAR. Assim que verificarmos que estas três rotinas estão corretas, iremos remover TESTE_IMP_HEXA de Es_VIDEO.ASM.

Crie a versão .COM de Es_video, e use o Debug para testar completamente IMP_HEXA. Mude o 3F na posição de memória 101h para cada um dos limites de condição que testamos no Capítulo 5, e depois use G para executar TESTE_IMP_HEXA a partir do Debug.

Usaremos muitos programas simples de teste para testar novas rotinas escritas. Deste modo, podemos montar um programa parte a parte, ao invés de tentar montá-lo e depurá-lo todo de uma vez. Este método incremental é muito mais rápido e fácil, pois podemos confinar os erros somente ao codigo novo, com os outros já testados e aprovados anteriormente.

Rudimentos do Projeto Modular

Note que, a frente de cada rotina em Es_video, incluímos um bloco de comentários descrevendo resumidamente a função de cada rotina. Ainda mais importante é que esses comentários dizem que registradores a rotina usa para passar informação de dentro para fora, e vice-versa, assim como também indica quais as outras rotinas usadas por ela. Como uma das características do método modular, o bloco de comentários permite-nos usar qualquer rotina somente olhando sua descrição. Não há necessidade de se estudar a rotina para entender o seu funcionamento. Isto também torna muito fácil a alteração em uma rotina sem que haja necessidade de se alterar as rotinas que a chamam.

Tambem usamos as instruções PUSH e POP para salvar e recuperar quaisquer registradores usados dentro da rotina. Faremos isso em todas as rotinas que escrevermos, exceto para nossas rotinas de teste. Este método também faz parte do estilo modular que estaremos usando.

Lembre-se de que nós salvamos e recuperamos quaisquer registradores usados para que nunca tenhamos que nos preocupar com interações complexas entre rotinas brigando por um espaço no grupo de registradores do 8088. Cada rotina é livre para usar tantos registradores quanto quiser, *desde que* os recupere antes da instrução RET. Este é um pequeno preço que se tem a pagar por conta da simplicidade. Além disso, sem salvar e recuperar registradores, a tarefa de reescrever as rotinas seria uma loucura. Tenha certeza de que você perderia muitos cabelos nesse processo.

Também tentamos usar muitas pequenas rotinas, ao invés de uma grande. Isso também torna a tarefa de programação mais simples, embora às vezes sejamos obrigados a escrever rotinas maiores quando o projeto se torna particularmente enrolado.

Estas idéias e métodos serão mais detalhadas nos capítulos a seguir. No próximo capitulo, iremos acrescentar uma outra rotina a Es_video: uma rotina para apanhar uma palavra no registrador DX e imprimir o número em decimal na tela.

Um Esqueleto de Programa

Como vimos neste capítulo e no anterior, o assembler impõe certas linhas adicionais a qualquer programa que escrevermos. Em outras palavras, precisamos escrever algumas pseudo-ops que dão informação básica ao assembler. Para referências futuras, eis o mínimo que você precisa para escrever seus programas:

```
CODE_SEG        SEGMENT
        ASSUME  CS:CODE_SEG
        ORG     100h

Alguma_rotina   PROC    NEAR



        INT     20h
Alguma_rotina   ENDP

CODE_SEG        ENDS
        END     Alguma_rotina
```

Nos capítulos seguintes, iremos acrescentar mais algumas pseudo-ops a este programa esqueleto, mas você pode usá-lo desse jeito como ponto de partida para novos programas que você escrever. Ou, melhor ainda, você pode usar alguns dos programas e rotinas deste livro como seu ponto de partida.

Resumo

Estamos realmente fazendo progresso agora. Neste capítulo, aprendemos a escrever rotinas em linguagem assembly. De agora em diante, as rotinas aparecerão o tempo todo, e usando rotinas menores os programas tornam-se mais facilmente utilizáveis.

Vimos que uma rotina começa com uma definição PROC e termina com uma pseudo-op ENDP. Reescrevemos IMPR_A_J para testar nosso novo conhecimento de rotinas, e depois passamos a reescrever nosso programa para que imprima um número hexa — desta vez com uma rotina extra. Agora que vimos a facilidade de se trabalhar com as rotinas, não há motivo para não as usarmos mais ainda. Na verdade, vimos que há muitas razões para se usar muitas pequenas rotinas.

Ao final deste capítulo, foi falado um pouco sobre o projeto modular, uma filosofia que nos economizará muito tempo e esforço. Nossos programas modulares serão mais fáceis de se escrever, mais fáceis de se ler, e mais fáceis de alguém modificar do que os programas criados com técnicas antiquadas de lógica "espagueti": programas escritos com longas rotinas e muitas interações.

Agora estamos prontos para montar uma outra rotina útil. Depois disso, no Capítulo 11, aprenderemos mais sobre os segmentos. E de lá, passaremos a desenvolver programas maiores, onde realmente começaremos a usar técnicas de projeto modular.

10

IMPRIMINDO EM DECIMAL

Relembrando a Conversão

Alguns Truques

O Funcionamento Interno

Resumo

Havíamos prometido uma rotina para pegar uma palavra e imprimi-la em notação decimal. IMP__DECIMAL usa alguns truques novos — modos de se encontrar um byte aqui, alguns microssegundos lá. Talvez esses truques não compensem o esforço aplicado. Mas se você os memorizar, descobrirá que pode usá-los para diminuir o tamanho e aumentar a velocidade dos programas. Através desses truques, também aprendemos alguns novos tipos de operações lógicas para acrescentar à instrução AND que vimos no Capítulo 5. Primeiro, vamos rever o processo de conversão de uma palavra para dígitos decimais.

Relembrando a Conversão

A divisão é a chave para se converter uma palavra em dígitos decimais. Lembre-se de que a instrução DIV calcula tanto a resposta inteira quanto o seu resto. Assim, o cálculo de 12345 / 10 gera 1234 como resposta inteira, e 5 como resto. Neste exemplo, 5 é simplesmente o dígito mais à direita. E se dividirmos por 10 novamente, teremos o próximo dígito à esquerda. A repetição sucessiva por 10 seleciona os dígitos da direita para a esquerda, colocando-o como resto a cada divisão.

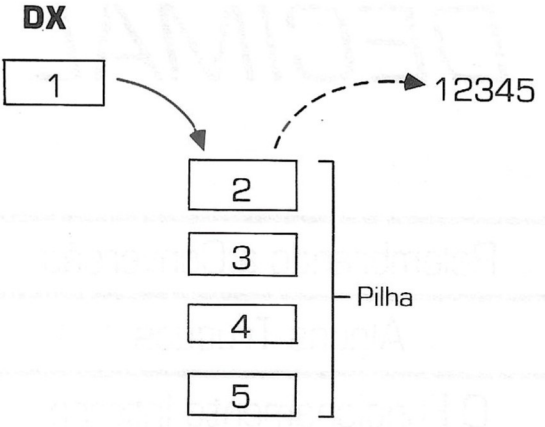


Figura 10-1. ***O Empilhamento de Dígitos Inverte a Sua Ordem.***

É claro que os dígitos vêm em sentido contrário, mas temos um modo de consertar isso na linguagem assembly. Lembra-se da pilha? Ela é exatamente como uma pilha de pratos: o primeiro a sair do topo da pilha foi o último a ser empilhado. Se substituirmos os pratos por dígitos, e colocarmos os dígitos um no topo do outro conforme aparecerem como resto, conseguiremos isso. Podemos retirar os dígitos na ordem correta.

O dígito do topo é o primeiro dígito do nosso número, e os outros dígitos estão abaixo dele. Dessa forma, se colocarmos os restos na pilha quando forem calculados, e retirarmos depois, um a um, para impressão, os dígitos virão na ordem correta.

O programa a seguir é uma rotina completa para imprimir um número em notação decimal. Como dissemos, há alguns truques escondidos na rotina. Logo voltaremos a eles, mas antes vamos tentar IMP__DECIMAL para ver se funciona antes de nos preocuparmos como isso é feito.

Coloque IMP__DECIMAL em ES__VIDEO.ASM, junto com as rotinas para imprimir um byte em hexa. Lembre-se de colocar IMP__DECIMAL *após* TESTE__IMP__HEXA, que será trocado para TESTE__IMP__DECIMAL. Para economizar algum trabalho, IMP__DECIMAL usa IMP__DIG__HEXA para converter um nibble (quatro bits) em um dígito.

Listagem 10-1. Acréscimo a ES_VIDEO.ASM

```
        PUBLIC      IMP_DECIMAL
;-------------------------------------------------------------------
; Esta rotina imprime um numero de 16 bits sem sinal em
; notacao decimal.
;
;       DX      N : numero de 16 bits sem sinal
;
; Usa:          IMP_DIG_HEXA
;-------------------------------------------------------------------
IMP_DECIMAL     PROC    NEAR
        PUSH    AX              ;Salva registradores usados
        PUSH    CX
        PUSH    DX
        PUSH    SI
        MOV     AX,DX
        MOV     SI,10           ;Dividira por 10 usando SI
        XOR     CX,CX           ;Contador digitos na pilha
NAO_ZERO:
        XOR     DX,DX           ;Zera palavra de N
        DIV     SI              ;Calcula N/10 e (N mod 10)
        PUSH    DX              ;Coloca um digito na pilha
        INC     CX              ;Mais um digito colocado
        OR      AX,AX           ;N ja igual a 0?
        JNE     NAO_ZERO        ;Nao, continua.
IMP_DIG_LOOP:
        POP     DX              ;Apanha os digitos em
                                ;   ordem contraria
        CALL    IMP_DIG_HEXA
        LOOP    IMP_DIG_LOOP
FIM_DECIMAL:
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
IMP_DECIMAL     ENDP
```

Note que incluímos um novo registrador, o SI (*Source Index — Índice Fonte*). Mais tarde veremos por que ele possui esse nome, e veremos seu irmão, o DI (*Destination Index — Índice Destino*). Ambos possuem usos especiais, mas também podem ser usados para qualquer propósito geral. Como IMP_DECIMAL precisa de quatro registradores de uso geral, usamos SI, mesmo que pudéssemos ter usado BX, simplesmente para mostrar que SI (e DI) podem servir como registradores de uso geral se precisarmos deles.

Antes de executarmos nossa nova rotina, precisamos fazer duas mudanças em ES_VIDEO.ASM. Primeiro, devemos remover a rotina TESTE_IMP_HEXA, e inserir esta rotina de teste em seu lugar:

Listagem 10-2. Troque TESTE_IMP_HEXA em ES_VIDEO.ASM por Esta Rotina

```
TESTE_IMP_DECIMAL       PROC    NEAR
    MOV         DX,12345
    CALL        IMP_DECIMAL
    INT         20h             ; Retorna ao DOS
TESTE_IMP_DECIMAL       ENDP
```

Esta rotina testa IMP_DECIMAL com o número 12345 (que o assembler converte para a palavra 3039h).

Segundo, precisamos mudar o comando END no final de ES_VIDEO.ASM para END TESTE_IMP_DECIMAL, pois nossa rotina principal agora é TESTE_IMP_DECIMAL.

Faça estas mudanças e dê uma rodada em ES_VIDEO. Converta-o para a versão .COM e veja se ele funciona. Se não funcionar, procure os erros no seu arquivo fonte. Se você tiver espírito de aventura, tente descobrir seu "bug" com o Debug. Afinal de contas, o Debug serve para isso.

Alguns Truques

Dentro de IMP_DECIMAL há dois truques do negócio, que poderão ser vistos novamente nos programas do ROM BIOS apresentados no Capítulo 17. O primeiro é uma eficiente instrução para se zerar um registrador. Não é muito mais eficiente do que um MOV AX,0, e talvez não valha o esforço, mas é o tipo de truque que as pessoas costumam usar. Vejamos a seguinte instrução:

```
XOR     AX,AX
```

Ela zera o registrador AX. Como? Para entender isso, precisamos ver uma outra operação lógica chamada OR Exclusivo, e daí o nome XOR.

O OR exclusivo é semelhante ao OR (que veremos a seguir), mas o resultado de XORar dois "verdadeiros":

XOR	0	1
0	0	1
1	1	0

é verdadeiro *somente* se um bit for verdadeiro, mas não se ambos forem verdadeiros. Assim, se operarmos com XOR um número consigo mesmo, obtemos zero:

```
      1011  0101
XOR   1011  0101
      ----  ----
      0000  0000
```

Esse é o truque. Não acharemos outros usos para a instrução XOR neste livro, mas creio que você o achará interessante.

Além desse, você verá algumas pessoas usando um outro truque rápido para zerar um registrador. Ao invés de usar a instrução XOR, poderíamos ter usado:

```
SUB     AX,AX
```

para zerar o registrador AX.

Agora vejamos o outro truque. Ele é um outro modo afastado de se zerar um registrador, e usa um primo do OR exclusivo — a função OR.

Vamos checar o registrador AX para ver se ele é zero. Para fazer isto, poderíamos usar a instrução CMP AX,0. Mas não, é preferível usar um truque: é mais divertido, e um pouco mais eficiente também. Assim, escrevemos OR AX,AX e seguimos esta instrução com um salto condicional JNE (Salta se Não Igual). (Poderíamos também usar JNZ — Salta se Não Zero.)

A instrução OR, como qualquer instrução matemática, seta os flags, incluindo o flag zero. Assim como AND, OR é um conceito lógico. Mas, aqui, o resultado é verdadeiro se um bit *OU* outro for verdadeiro:

OR	0	1
0	0	1
1	1	1

Se pegarmos um número e realizarmos um OR consigo mesmo, obteremos o número original de volta:

```
      1011  0101
OR    1011  0101
      1011  0101
```

A instrução OR também é útil para setar somente um bit em um byte. Por exemplo, podemos setar o bit 3 do número que acabamos de usar:

```
      1011  0101
OR    0000  1000
      1011  1101
```

Temos alguns outros truques que vamos jogar no decorrer deste livro, mas estes dois são os únicos que são inteiramente usados por diversão.

O Funcionamento Interno

Para ver como IMP_DECIMAL realiza seu trabalho, estude a listagem; não cobriremos mais detalhes aqui. Precisamos descobrir algumas coisas novas.

Primeiro, o registrador CX é usado para contar quantos dígitos colocamos na pilha, para que saibamos quantos remover. O registrador CX é uma escolha particularmente conveniente, pois podemos fazer um loop com a instrução LOOP e usar o registrador CX para controlar a contagem das repetições. Nossa escolha torna o loop de saída de dígito (IMP_DIG_LOOP) quase trivial, pois a instrução LOOP usa o registrador CX diretamente. Usaremos muito o registrador CX quando tivermos que armazenar um contador.

A seguir, tenha o cuidado de checar os limites de condição aqui. O limite 0 não é problema, como você pode ver. O outro limite é 65535, ou FFFFh, que você pode checar facilmente com o Debug. Basta carregar ES_VIDEO.COM para o Debug digitando *DEBUG ES_VIDEO.COM* e alterar os números 12345 (3039H) em 101h para 65535 (FFFFh). (IMP_DECIMAL trabalha com números sem sinal. Veja se você pode escrever uma versão para imprimir números com sinal.)

Você deve ter notado algo estranho aqui, tendo a ver com o 8088, e não com o nosso programa. O Debug trabalha quase sempre com bytes (pelo menos, o comando E) mas queremos alterar uma palavra. Devemos ter cuidado, pois o 8088 armazena os bytes em uma ordem diferente. Aqui está uma versão desassemblada para a instrução MOV:

```
3985:0100 BA3930        MOV     DX,3039
```

Você pode ver por *BA3930* que o byte em 101h é 39, e o que está em 102h é 30h (BA é o código da instrução MOV). Os dois bytes são os dois bytes de 3039h, mas aparentemente em ordem inversa. Parece confuso? Na verdade, a ordem torna-se lógica após uma pequena explicação.

Uma *palavra* consiste em duas partes, o byte baixo e o byte alto. O byte baixo é o byte menos significativo (39h em 3039h), enquanto o byte alto é a outra parte (30h). Faz sentido, então, colocar o byte mais baixo no endereço mais baixo na memória. (Alguns computadores realmente invertem estes dois bytes, e isto pode ficar meio confuso se você usar diversos computadores diferentes.)

Tente números diferentes para a palavra que se inicia em 101h, e então você verá como esse armazenamento funciona. Use TESTE_IMP_DECIMAL para ver se está tudo certo, ou então desassemble a primeira instrução.

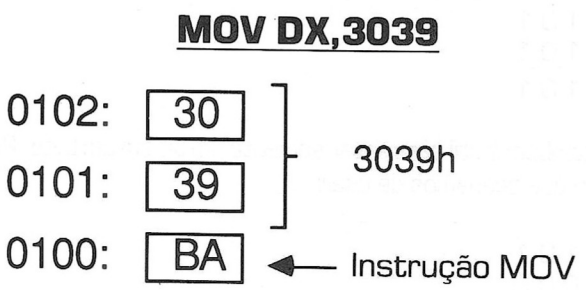


Figura 10-2. ***O 8088 Armazena os Números Com o Byte Mais Baixo Primeiro na Memória.***

Resumo

Neste capítulo pudemos acrescentar algumas instruções novas ao nosso repertório, bem como alguns truques. Também aprendemos algo sobre dois novos registradores, SI e DI, que podemos usar sobre registradores de uso geral. Eles também possuem outras finalidades, como veremos nos capítulos seguintes.

Aprendemos sobre as instruções lógicas XOR e OR, que nos permitem trabalhar com bits individuais de dois bytes ou palavras. E em nossa rotina IMP_DECIMAL, usamos a instrução XOR AX,AX como um truque para zerar o registrador AX. Usamos OR AX,AX como um outro modo de se escrever o equivalente de CMP AX,0, testando o registrador AX e vendo se ele é zero.

Finalmente, aprendemos sobre como o 8088 armazena uma palavra na memória, checando os limites de condição para nossa nova rotina, IMP_DECIMAL.

Aqui, ao final deste capítulo, já temos outra rotina de uso geral, IMP_DECIMAL, que poderemos usar no futuro para nossos próprios programas.

Respire um pouco agora. Temos alguns capítulos *diferentes* planejados a seguir. O Capítulo 11 cobre em detalhes o assunto de segmentos. Os segmentos, provavelmente, são a parte mais complicada do microprocessador 8088, de modo que o capítulo pode parecer um pouco pesado. Mesmo assim, no entanto, precisamos ver este assunto para podermos prosseguir.

Após isso, faremos uma pequena correção no curso e voltaremos ao nosso rumo aprendendo sobre o que deve ser o programa Dskpatch. Faremos alguns testes com os discos, aprendendo sobre setores, trilhas, e coisas semelhantes.

Daqui por diante, podemos planejar um rumo simples para as versões preliminares de Dskpatch. Enquanto isso, você aprenderá como se desenvolve grandes programas. Os programadores não escrevem programas inteiros, e depois o testam. Eles escrevem seções e testam cada uma antes de seguir adiante — o trabalho é muito menor desta forma. Usamos este método, de certa forma, escrevendo e testando IMP_HEXA e IMP_DECIMAL, para os quais os programas de teste foram muito simples. Para testar os programas daqui por diante, a tarefa será mais complexa, mas também mais interessante.

11

SEGMENTOS

Seccionando a Memória do 8088

Pseudo-Ops de Segmento

A Pseudo-Op ASSUME

CALLs Próximos e Distantes

Mais Sobre Instruções INT

Vetores de Interrupção

Resumo

Nos capítulos anteriores, encontramos diversas pseudo-ops que lidavam com segmentos. Agora é hora de vermos os próprios segmentos, e como o 8088 consegue endereçar todo um megabyte de memória (1.048.576 bytes). Depois disso, veremos por que os segmentos precisam de suas próprias pseudo-ops no assembler, e nos próximos capítulos começaremos a usar diferentes segmentos (até aqui, temos usado somente um). Depois, no Capítulo 13, aprenderemos sobre o projeto modular, e veremos como agrupar os segmentos em um arquivo .COM.

Vamos começar a nível do 8088, aprendendo como construir um endereço de 20 bits necessário para endereçar um megabyte completo de memória.

Seccionando a Memória do 8088

Os segmentos são a única parte do 8088 que ainda não explicamos, e talvez seja a parte mais confusa do microprocessador para a maior parte das pessoas. De fato, os segmentos são o que existe de mais *enrolado* no negócio: ele é um truque para resolver um sério problema.

O problema, neste caso, é ser capaz de endereçar mais de 64K de memória — o limite de uma palavra, pois 65535 é o maior número que uma palavra pode conter. A Intel, criadora do 8088, usou segmentos e registradores de segmentos para "consertar" esse problema, e com isso tornou o 8088 mais confuso.

Até aqui, ainda não nos preocupamos com esse problema. Usávamos o registrador IP para conter o endereço da próxima instrução que o 8088 deve executar desde que encontramos o Debug no Capítulo 2. Você deve lembrar-se de que nós dissemos que o endereço era formado por um valor no registrador CS e outro no registrador IP. Mas nunca realmente dissemos isso. Vamos descobrir isso agora.

Embora o endereço completo seja formado por dois registradores, o 8088 não forma um número de duas palavras para o endereço. Se você fosse usar os registradores CS:IP como um número de 32 bits (dois números de 16 bits lado a lado), o 8088 seria capaz de endereçar cerca de quatro bilhões de bytes, que é muito mais do que um milhão de bytes que ele realmente endereça. O método do 8088 é um pouco mais complicado; o registrador CS fornece o endereço *inicial* para o segmento de código, onde um segmento significa 64K de memória. Veja como isso funciona.

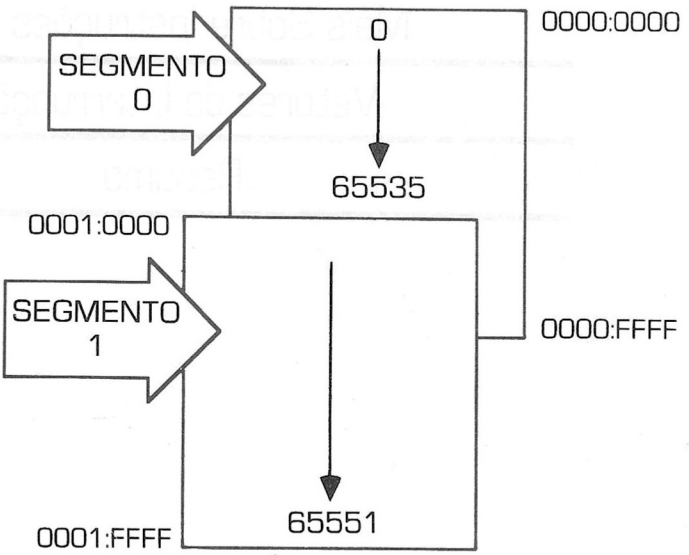


Figura 11-1. ***A Superposição de Segmentos Começa a Cada 16 Bytes, e se Estende por 65536 Bytes.***

Como você pode ver na Figura 11-1, o 8088 divide a memória em muitos segmentos superpostos, com um novo segmento começando a cada 16 bytes. O primeiro segmento (segmento 0) começa na posição de memória 0; o segundo (segmento 1), começa em 10h (16); o terceiro começa em 20h (32), e assim por diante.

O endereço real é exatamente CS X 16 + IP. Por exemplo, se o registrador CS contém 3FA8, o endereço absoluto será:

```
CS X 16 : 0011  1111  1010  1000  0000
    IP :        1101  0000  0001  0111
        0100  1100  1010  1001  0111
```

se IP for D017 . Multiplicamos por 16 para deslocar CS quatro bits à esquerda, acrescentando zeros à direita.

Isso pode parecer um modo estranho de se endereçar mais do que 64K de memória — e é, mas funciona. Logo começaremos a ver como ele funciona bem.

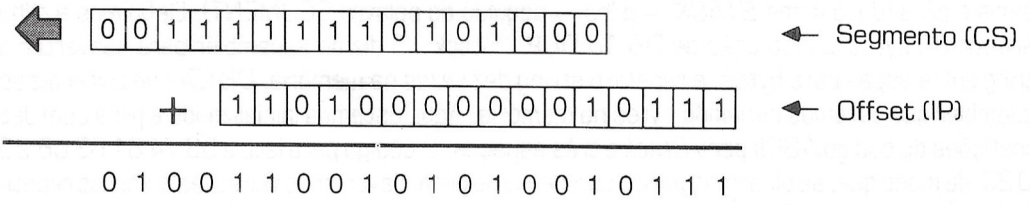


Figura 11-2. ***O Endereço Absoluto de CS:IP é CS X 16 + IP.***

Na verdade, o 8088 possui quatro registradores de segmento: o CS (*Code Segment — Segmento de Código*), o DS (*Data Segment — Segmento de Dados*), o SS (*Stack Segment — Segmento de Pilha*), e o ES (*Extra Segment — Segmento Extra*). Já vimos que o registrador CS é usado para indicar ao 8088 o segmento onde se encontra a próxima instrução a ser executada. Da mesma forma, o registrador DS é o segmento onde o 8088 procura seus dados, e o SS é o segmento onde o 8088 procura a pilha.

Antes de continuarmos, vejamos um pequeno programa, bem diferente dos que já vimos antes, usando dois segmentos diferentes. Entra este programa no arquivo TEST_SEG.ASM:

Listagem 11-1. O Programa TEST_SEG.ASM

```
CODE_SEG            SEGMENT
          ASSUME    CS:CODE_SEG
TEST_SEGMENT        PROC    NEAR
          MOV       AH,4Ch              ;Ask for the exit-to-dos function
          INT       21h                 ;Return to DOS
TEST_SEGMENT        ENDP
CODE_SEG            ENDS

STACK_SEGMENT       SEGMENT STACK
          DB        10 DUP ("Stack  ")  ;Three spaces after Stack
STACK_SEGMENT       ENDS

          END       TEST_SEGMENT
```

Depois monte (com o assembler) e linkedite o programa, mas não gere um arquivo .COM para ele. O resultado será TEST_SEG.EXE, que é um pouco diferente de um arquivo .COM.

Nota: Temos que usar um método diferente para sair dos arquivos .EXE. Para arquivos .COM, INT 20 funciona perfeitamente bem, mas não funciona em arquivos .EXE porque a organização dos segmentos é muito diferente, como veremos neste capítulo; veja mais sobre isso adiante.

Quando usamos o Debug em um arquivo .COM, o Debug coloca todos os registradores de segmento com o mesmo número, com o programa começando no *deslocamento* (*offset*) 100h a partir do início do segmento. Os primeiros 256 bytes (100h) são usados para armazenarem várias informações em que não estamos interessados, mas logo veremos algo sobre esta área.

Agora, tente carregar TEST_SEG.EXE para o Debug, para ver o que acontece com os segmentos em um arquivo .EXE:

```
A>DEBUG TEST_SEG.EXE
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=0050 BP=0000 SI=0000 DI=0000
DS=3985 ES=3985 SS=3996 CS=3995 IP=0000   NV UP DI PL NZ NA PO NC
3995:0000 CD20              INT         20
```

Os valores dos registradores SS e CS são diferentes dos encontrados em DS e ES.

Em nosso programa, definimos dois segmentos. O STACK_SEGMENT é o local onde colocamos a pilha (daí o nome *STACK* — pilha — seguido da palavra SEGMENT). Definimos a pilha para ter 80 bytes: a instrução de DB 10 DUP ("Stack ") diz ao assembler para converter o string entre aspas para bytes, e repetir o string dez vezes na memória. DB (*Define Byte*) diz ao assembler que estamos definindo bytes na memória. Aqui, estamos inicializando a pilha com dez repetições do código ASCII para *Stack* e três espaços. O código para isto é 53 74 61 63 6B 20 20 20, de modo que, se olharmos para o segmento de pilha, deveremos ver estes números repetidos dez vezes. Peça ao Debug para copiar esta área de memória com o comando a seguir, que diz ao Debug para copiar a memória começando no deslocamento 0 dentro do Segmento de Pilha (SS:0):

```
-D SS:0
3996:0000  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20  Stack   Stack
3996:0010  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20  Stack   Stack
3996:0020  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20  Stack   Stack
3996:0030  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20  Stack   Stack
3996:0040  53 74 61 63 6B 20 20 20-53 74 61 63 6B 20 20 20  Stack   Stack ..
3996:0050  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

SS SP

O endereço do topo da pilha, dado por SS:SP. SP é o *"Stack Pointer"* (Ponteiro de Pilha), semelhante ao IP para o CS, e é o deslocamento dentro do Segmento de Pilha Atual.

Na verdade, a expressão "topo-da-pilha" não é muito correta, pois a pilha cresce a partir da memória alta, em direção à parte baixa. Assim, o topo da pilha está no fundo da pilha, e as novas entradas são colocadas progressivamente nos endereços menores. Aqui, SP é 50h, que é 80 decimal, pois definimos uma pilha com 80 bytes de extensão. Não colocamos nada na pilha ainda, de modo que o topo da pilha ainda é o topo da memória que reservamos para a pilha: 50h.

Agora que você já sabe como encontrar a pilha, você pode ver como ela se altera nos programas anteriores. Mas, agora, vamos continuar com o exemplo já no Debug.

Note que o Segmento de Pilha (SS) é o segmento número 3996 (provavelmente não será o mesmo para o seu computador), enquanto nosso segmento de código é 3995 — um a menos do que SS, ou seja, 16 bytes a menos na memória. Isto significa que se gerarmos uma listagem desassemblada na memória começando em CS:0, veremos nosso programa (a instrução INT 20) seguida de 14 bytes iguais a zero (o INT 20 ocupa dois bytes), e depois veremos os bytes do segmento de pilha.

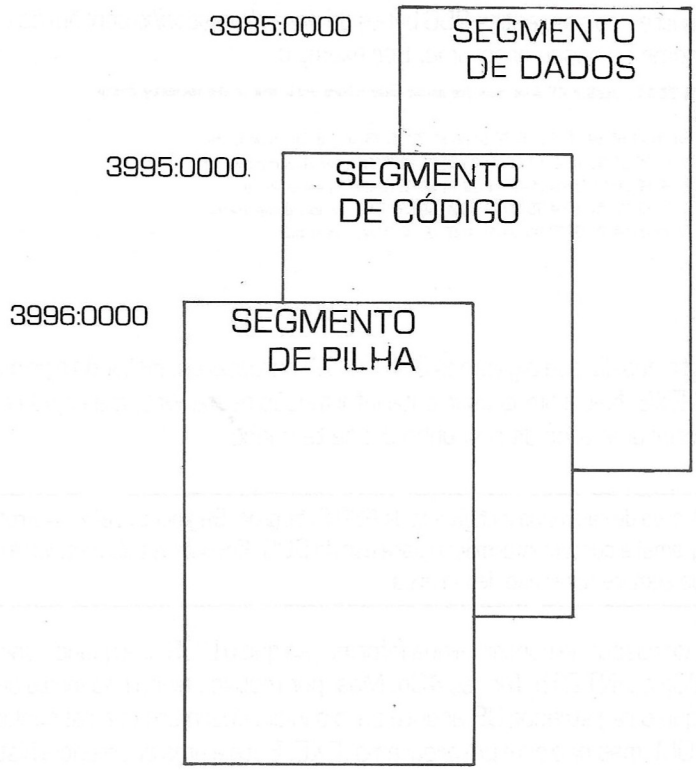


Figura 11-3. ***Esboço da Memória para TEST_SEG.EXE.***

Também veremos os dados de *Stack* seguidos por três espaços, desassemblados:

```
-U CS:0
3995:0000 CD20      INT     20
3995:0002 0000      ADD     [BX+SI],AL
3995:0004 0000      ADD     [BX+SI],AL
3995:0006 0000      ADD     [BX+SI],AL
3995:0008 0000      ADD     [BX+SI],AL
3995:000A 0000      ADD     [BX+SI],AL
3995:000C 0000      ADD     [BX+SI],AL
3995:000E 0000      ADD     [BX+SI],AL
3995:0010 53        PUSH    BX
3995:0011 7461      JZ      0074
3995:0013 63        DB      63
3995:0014 6B        DB      6B
3995:0015 2020      AND     [BX+SI],AH
3995:0017 205374    AND     [BP+DI+74],DL
3995:001A 61        DB      61
3995:001B 63        DB      63
3995:001C 6B        DB      6B
3995:001D 2020      AND     [BX+SI],AH
3995:001F 205374    AND     [BP+SI+74],DL
```

Assim como era esperado, o número 53h — o código ASCII de S, a primeira letra em nossa área de pilha — está no deslocamento 10h (16) dentro do nosso segmento de código.

Olhando para os registradores, você pode notar que os registradores ES e DS contêm 3985h, 10h a menos que o início do programa no segmento 3995. Multiplicando por 16 para obter o número de bytes, vemos que há 100h (256) bytes antes do início do programa. Esta é a mesma área colocada antes do início de um arquivo .COM.

Entre outras coisas, estes 256 bytes de *área de rascunho* contêm os caracteres que digitamos após o nome do nosso programa. Por exemplo:

```
A>DEBUG TEST__SEG.EXE And now for some characters we'll see in the memory dump
-D DS:80
3985:0080 39 20 41 6E 64 20 6E 6F-77 20 66 6F 72 20 73 6F  9 And now for so
3985:0090 6D 65 20 63 68 61 72 61-63 74 65 72 73 20 77 65  me characters we
3985:009A 27 6C 6C 20 73 65 65 20-69 6E 20 74 68 65 20 6D  'll see in the m
3985:00B0 65 6D 6F 72 79 20 64 75-6D 70 0D 20 6D 65 6D 6F  emory dump. memo
3985:00C0 72 79 20 64 75 6D 70 0D-00 00 00 00 00 00 00 00  ry dump.........
```

O primeiro byte nos diz que digitamos 39h (ou 57) caracteres, incluindo o primeiro espaço logo após TEST__SEG.EXE. Não iremos usar esta informação neste livro, mas ajuda a mostrar por que poderíamos querer uma área de rascunho desse tamanho.

Nota: A área de rascunho é chamada de PSP ("Program Segment Prefix" — Prefixo do Segmento de Programa) e contém informação para uso do DOS. Em outras palavras, você não deve assumir que pode sempre fazer uso dessa área.

A área de rascunho contém ainda informação que o DOS usa quando saímos de um programa, ou com INT 20h ou INT 21h, função 4Ch. Mas, por motivos ainda não muito claros, a instrução INT 20h espera que o registrador CS aponte para o início desta área de rascunho, que ele faz com um programa .COM, mas *não* com um programa .EXE. Esta é uma ocorrência histórica. De fato, a função de saída (função 4Ch de INT 21h) foi introduzida na versão 2.0 do DOS.

O código para arquivos .COM sempre deve começar no endereço 100h no segmento de código para deixar espaço para essa "área de rascunho". Isso não acontece em arquivos .EXE, onde o código inicia em IP = 0000, pois o segmento de código iniciou 100h bytes após o início da área na memória.

Lembre-se que, em nossos arquivos .COM no Capítulo 10, tínhamos colocado explicitamente uma pseudo-op ORG 100h no início dos nossos programas para reservar 100h bytes. A pseudo-op ORG 100h define a *origem* do nosso código em 100h. Isso é tudo que ela faz, mas continuaremos a usar ORG 100h em nossos arquivos.

Mostramos um programa .EXE aqui só para que você pudesse aprender sobre segmentos. Mais adiante, aprenderemos mais sobre eles, mas usaremos arquivos .COM daqui por diante, pois eles são menores e são carregados mais rapidamente na memória. Você verá as razões para isto quando chegarmos ao último capítulo, mas agora vamos em frente. Vamos aprender sobre as pseudo-ops para os segmentos.

Pseudo-Ops de Segmento

Temos diversas pseudo-ops para estudar: SEGMENT, ENDS, ASSUME, e o NEAR e FAR para a pseudo-op PROC. Também precisamos dar uma olhada atenta nas instruções CALL e RET. Quando vimos esta base, aprendemos sobre a instrução INT e vimos a sua semelhança com uma instrução CALL. Mas vamos colocar uma ordem aqui, e começar com SEGMENT e ENDS.

As pseudo-ops SEGMENT e ENDS são semelhantes às pseudo-ops PROC e ENDP que encontramos no Capítulo 9. Definimos um segmento limitando parte do código fonte com o par SEG-

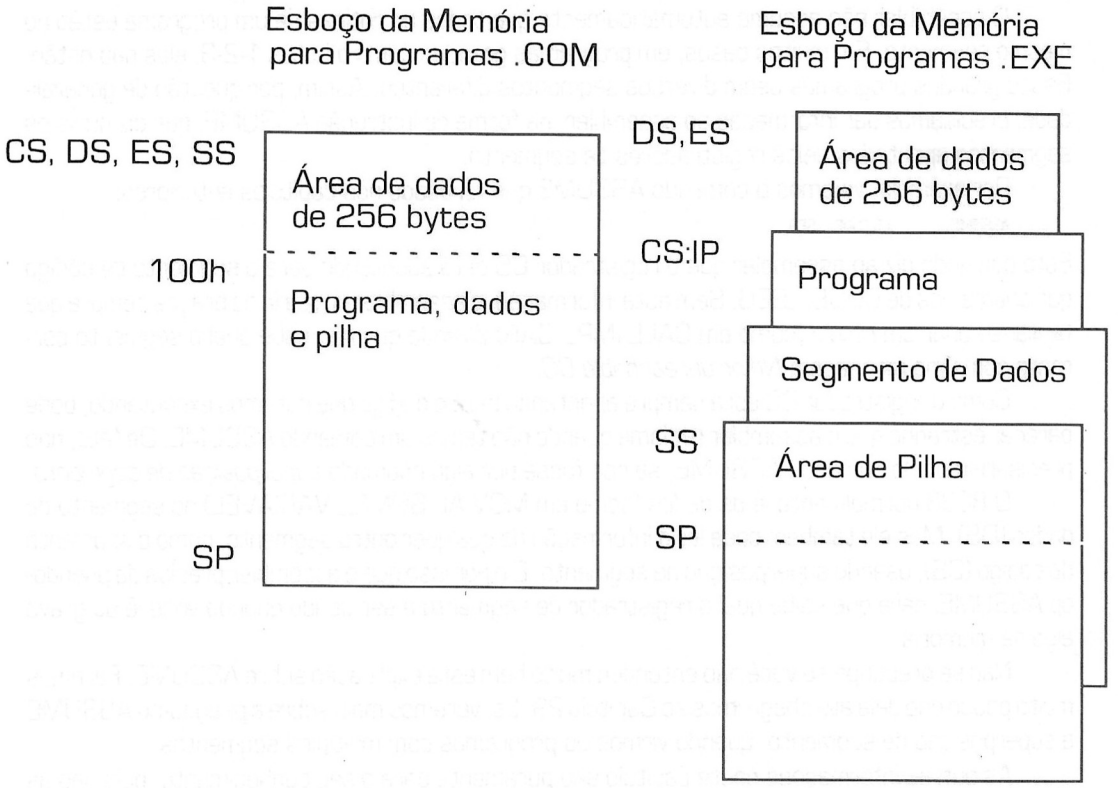


Figura 11-4. ***Programas .COM e .EXE.***

MENT/ENDS, do mesmo modo com que definimos uma sub-rotina com o par PROC/ENDP. O nome que aparece antes da pseudo-op SEGMENT é um rótulo.

Usaremos este rótulo no Capítulo 13, quando dividiremos nosso arquivo fonte em diversos arquivos fonte diferentes e dois segmentos; um segmento de dados e um segmento de código. Com dois segmentos, podemos separar facilmente as variáveis na memória em nosso programa. Haverá mais sobre variáveis também no Capítulo 13, e ainda acrescentaremos mais partes ao pseudo-op SEGMENT. Há, no entanto, milhares de detalhes em que não perderemos tempo. Você pode encontrar informações em seu manual de assembler, se assim desejar.

A Pseudo-Op ASSUME

A pseudo-op ASSUME é um pouco diferente de SEGMENT. Ela dá informação ao assembler sobre os segmentos, e como queremos usar os registradores de segmento. Para entender ASSUME, precisamos antes entender como o assembler lida com os rótulos e nomes de variáveis.

Toda vez que você cria um rótulo, como uma rotina (como IMP_CAR PROC NEAR) ou uma variável na memória, o assembler lida com diversas informações além do nome: o tipo (rotina, byte, palavra etc.), o endereço do nome, e o segmento no qual ele está definido. O ASSUME está envolvido com essa última informação.

O assembler não assume automaticamente que todas as rotinas de um programa estão no mesmo segmento. Em muitos casos, em programas grandes como o Lotus 1-2-3, elas não estão. Esses grandes programas usam diversos segmentos diferentes. Assim, por questão de generalidade, precisamos dar informação ao assembler, na forma de instrução ASSUME, que diz quais os segmentos apontados pelos registradores de segmento.

Por exemplo, vejamos o comando ASSUME que foi usado nos capítulos anteriores:

```
ASSUME      CS:CODE_SEG
```

Este comando diz ao assembler que o registrador CS está apontando para o segmento de código que chamamos de CODE_SEG. Sem esta informação, o assembler cruzaria os braços sempre que tentasse usar um rótulo (como em CALL IMP_CAR) dizendo que não sabe qual o segmento corrente com uma mensagem *No or unreachable CS*.

Como o registrador CS está sempre apontando para o código que estamos executando, pode parecer estranho que o assembler reclame quando não temos um comando ASSUME. De fato, não precisaríamos do comando ASSUME, se não fosse por algo chamado *superposição de segmento*.

O 8088 normalmente lê os dados (como em MOV AL,SOME_VARIAVEL) no segmento de dados (DS). Mas ele também pode ler a informação de qualquer outro segmento, como o segmento de código (CS), usando superposição de segmento. E é por isso que o assembler precisa da pseudo-op ASSUME: para que saiba qual o registrador de segmento a ser usado quando você lê ou grava algo na memória.

Não se preocupe se você não entendeu muito bem esta explicação sobre ASSUME. Faremos muito pouco uso dela até chegarmos ao Capítulo 29. Lá, veremos mais sobre a pseudo-op ASSUME e superposição de segmento, quando virmos os programas com múltiplos segmentos.

As outras informações neste capítulo são puramente para o seu conhecimento, pois não as utilizaremos neste livro. Você poderá saltar as duas seções seguintes e voltar a elas mais tarde, isto se você achar o assunto difícil ou estiver ansioso para voltar à programação.

CALLs Próximos e Distantes

Vamos voltar um pouco no tempo e lembrar das instruções CALL que usamos nos capítulos anteriores. Especificamente, vejamos um pequeno programa do Capítulo 7, onde aprendemos sobre a instrução CALL. Naquela ocasião escrevemos um pequeno programa, que se parece assim (sem a rotina em 200h):

```
3985:0100 B241      MOV     DL,41
3985:0102 B90A00    MOV     CX,000A
3985:0105 E8F800    CALL    0200
3985:0108 E2FB      LOOP    0105
3985:010A CD20      INT     20
```

Você pode ver, olhando o código de máquina à esquerda, que a instrução CALL ocupa somente três bytes (E8F800). O primeiro byte (E8h) é a instrução CALL, e os dois bytes seguintes compõem um deslocamento. O 8088 calcula o endereço da rotina que estamos chamando, somando esse deslocamento 00F8h (lembre-se de que o 8088 armazena o byte mais baixo em primeiro lugar, seguido pelo byte mais alto, em ordem contrária à que usamos normalmente para reapresentar um endereço) ao endereço da próxima instrução (108h em nosso programa). Neste caso, então, temos F8h + 108h = 200h. Exatamente o que esperávamos.

O fato de que esta instrução só usa uma única palavra para o deslocamento significa que os CALLs estão limitados a um único segmento, que possui 64K bytes. Mas, como podemos escrever programas como o Lotus 1-2-3, que é bem maior do que 64K? Fazemos isso usando chamadas FAR (distantes) ao invés de NEAR (próximas).

CALLs NEAR, como vimos, são limitados a um único segmento. Em outras palavras, eles alteram o registrador IP sem alterar o registrador CS. E por esta razão, elas são às vezes conhecidas como CALLs (ou chamadas) *intra-segmento*.

Mas também podemos ter chamadas FAR, que alteram tanto CS quanto IP. Tais chamadas geralmente são conhecidas como *inter-segmento*, pois elas chamam rotinas que se encontram em outros segmentos.

Em conseqüência destas duas versões da instrução CALL, há duas versões da instrução RET.

O CALL NEAR, como vimos no Capítulo 7, coloca uma única palavra na pilha, que é o seu endereço (deslocamento) de retorno. E a instrução RET correspondente retira esta palavra da pilha, e a retorna ao registrador IP.

No caso dos CALLs e RETs de uma chamada FAR, uma palavra não é suficiente, pois estamos trabalhando em outro segmento. Em outras palavras, precisamos salvar um endereço de duas palavras na pilha: uma palavra para o ponteiro de instrução (IP) e a outra para o segmento de código (CS). O FAR RET, portanto, retira duas palavras da pilha — colocando uma no registrador CS e outra no IP.

Agora, um fato interessante. Como o assembler sabe qual destes CALLs e RETs usar? Quando ele deve usar o FAR CALL e quando deve usar o NEAR CALL? É aí que as pseudo-ops NEAR e FAR tomam o comando.

Como exemplo, veja o seguinte programa:

```
PROC_UM         PROC        FAR

                .
                .
                .
                RET
PROC_UM         CALL        PROC_UM     NEAR

                .
                .
                .
                RET
PROC_DOIS       ENDP
```

Quando o assembler vê a instrução CALL PROC_UM, ele procura em sua tabela a definição de PROC_UM, que, neste caso, é PROC_UM PROC FAR. Esta definição diz se a rotina é próxima ou distante (NEAR ou FAR).

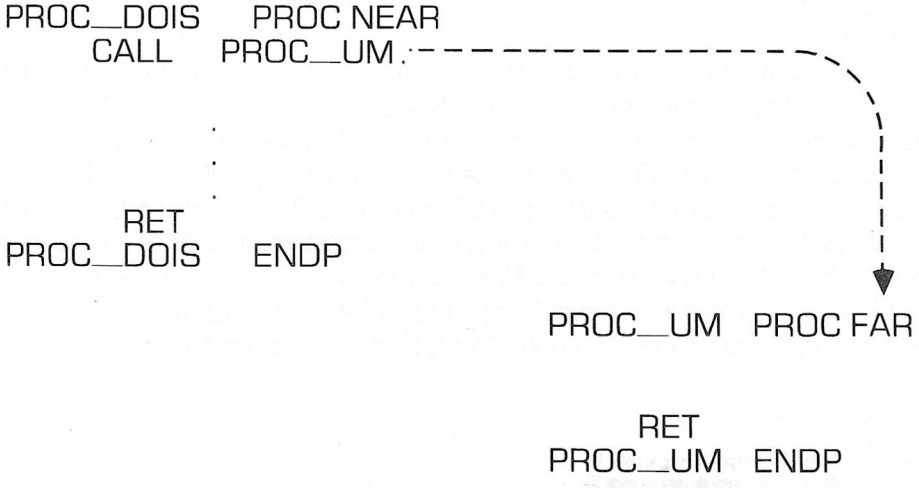


Figura 11-5. ***O Assembler Produz um CALL FAR.***

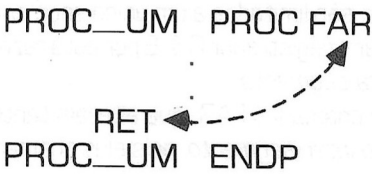


Figura 11-6. ***O Assembler Produz um RET FAR.***

No caso de uma rotina NEAR, o assembler gera uma chamada (CALL) próxima. Por outro lado, ele gera um CALL distante se a rotina for FAR. Em outras palavras, o assembler usa a definição da rotina que você está *chamando* para determinar o tipo de CALL necessário.

Para a instrução RET, o assembler olha a definição da rotina que contém a instrução RET. Em nosso programa, a instrução RET para PROC_UM será um FAR RET, pois PROC_UM está sendo declarada como uma rotina FAR. Entretanto, o RET em PROC_DOIS é um NEAR RET.

Mais Sobre Instruções INT

A instrução INT é semelhante à instrução CALL, mas com uma pequena diferença. O nome INT vem da palavra *interrupção*. Uma interrupção é um sinal externo que faz com que o 8088 execute uma rotina e depois retorne ao que estava fazendo antes de ter recebido a interrupção. Uma instrução INT não interrompe o 8088, mas é tratada como se o fizesse.

Quando o 8088 recebe uma interrupção, ele precisa armazenar mais informação na pilha do que simplesmente as duas palavras do endereço de retorno. Ele tem que armazenar os valores dos flags de status — o flag de carry, de zero etc. Estes valores são armazenados em uma palavra conhecida como Registrador de Flag, e o 8088 coloca essa informação na pilha antes do endereço de retorno. É por isso que precisamos salvar os flags de status.

O seu IBM PC normalmente responde a uma série de interrupções diferentes. O 8088 dentro do PC recebe uma interrupção do clock 18,2 vezes a cada segundo, por exemplo. Cada uma dessas interrupções faz com que o 8088 pare de fazer o que está fazendo e execute uma rotina para contar os pulsos de clock.

Agora, imagine uma interrupção acontecendo entre estas duas instruções de programa:

```
CMP     AH,2
JNE     NAO_2
```

Vamos supor que AH = 2, de modo que o flag zero será setado após a instrução CMP, o que significa que a instrução JNE não deverá causar um desvio para NAO_2.

Suponha agora que o 8088 receba uma interrupção no meio das duas instruções, significando que ela será atendida antes que o 8088 cheque o flag de zero (com a instrução JNE). Se o 8088 não salvasse e recuperasse os registradores de flag, o 8088 usaria os flags setados pela rotina de interrupção, e *não* os da nossa instrução CMP. Para prevenir desastres, o 8088 *sempre* salva e recupera o registrador de flag nas interrupções. Uma interrupção salva os flags, e uma instrução IRET (*Interrupt Return — retorno de interrupção*) restaura os flags ao final da rotina.

O mesmo acontece para uma instrução INT. Assim, após executar a instrução:

```
INT     21
```

a pilha do 8088 fica assim:

```
Topo da Pilha —>   Antigo IP (endereço de retorno parte I)
                   Antigo CS (endereço de retorno parte II)
                   Antigo Registrador de Flags
```

(A pilha cresce em direção à parte mais baixa da memória, de modo que o topo-da-pilha esta abaixo do Antigo Registrador de Flags.)

Quando colocamos uma instrução INT em um programa, contudo, a interrupção não é surpresa. Por que, então, vamos querer os flags salvos? Isso não é útil somente quando temos uma interrupção imprevisível? A resposta, na verdade, é não. Há uma razão muito boa para se salvar e recuperar os flags para instruções INT. De fato, sem esta característica, o Debug não seria possível.

O Debug usa um flag especial no registrador de flag chamado de Flag de "Trap". Este flag coloca o 8088 em um modo especial conhecido como modo *passo-a-passo*, que o Debug usa para acompanhar os programas uma instrução de cada vez. Quando o flag de Trap é setado, o 8088 emite um INT 1 após executar qualquer instrução.

O INT 1 também zera o flag de trap, de modo que o 8088 não continue a executar passo-a-passo enquanto estiver executando a rotina de INT 1 do Debug. Mas como INT 1 salvou os flags para a pilha, a emissão de um IRET para o programa que estamos depurando recupera o flag de trap. Depois, receberemos um outro INT 1 após a próxima instrução no programa. Este é só um exemplo da utilidade de se salvar os flags. Porém, como veremos adiante, esta característica de recuperar os flags nem sempre é apropriada.

Algumas rotinas de interrupção "passam por cima" da recuperação dos registradores de flag. Por exemplo, às vezes INT 21 altera os registradores de flag, alterando o processo normal de retorno. Muitas das rotinas INT 21h que lêem ou gravam informação em disco retornam com o flag de carry setado, se houve um erro de algum tipo (por exemplo, se não houver disco no drive).

Vetores de Interrupção

Onde essas instruções de interrupção apanham os endereços de suas rotinas? Cada instrução de interrupção possui um número de interrupção, tal como 21h em INT 21h. O 8088 descobre os endereços para as rotinas de interrupção em uma tabela de *vetores de interrupção*, que está localizada bem no início da memória. Por exemplo, o endereço de duas palavras para a rotina de INT 21h está em 0000:0084. Obtemos esse endereço, multiplicando o número da interrupção por 4 (4 x 21h = 84h), pois precisamos de quatro bytes (duas palavras) para cada vetor ou endereço de rotina.

Esses vetores são muitíssimos úteis para o acréscimo de características ao DOS, pois nos capacitam a interceptar chamadas para rotinas de interrupção, alterando os endereços na tabela de vetor. No entanto não faremos isso neste livro. Alguns truques são muito avançados para nós agora.

Todas estas idéias e métodos devem tornar-se mais claros quando virmos mais exemplos. A maioria deste livro, daqui por diante, estará cheia de exemplos, de modo que teremos muito a estudar. Você deve estar se sentindo meio confuso com tanta informação nova, portanto descanse um pouco. Tomaremos mais um pouco de fôlego no próximo capítulo, para nos reorientarmos de volta ao curso normal.

Resumo

Como dissemos, este capítulo conteve muita informação. Não usaremos tudo isso, mas precisávamos aprender mais sobre segmentos. O Capítulo 13 nos trará de volta ao projeto modular, e usaremos alguns aspectos dos segmentos para tornar nossa tarefa mais fácil.

Começamos este capítulo aprendendo como o 8088 divide a memória em segmentos. Para entender os segmentos em mais detalhes, montamos um programa .EXE com dois segmentos di-

ferentes. Não usaremos os programas .EXE neste livro, mas um programa assim pode demonstrar muito bem a idéia dos segmentos.

Também descobrimos que a área de rascunho de 100h (256) bytes antes do nosso programa contém uma cópia do que digitarmos na linha de comando. Novamente, não usaremos este conhecimento neste livro, mas é importante sabermos por que o DOS reserva este trecho tão grande de memória.

E, finalmente, passamos a explicar sobre as pseudo-ops SEGMENT, ENDS, ASSUME, NEAR e FAR. Todas elas são pseudo-ops que nos ajudam a trabalhar com segmentos. Neste livro, usaremos pouco a potência destas instruções, pois nossos programas .COM só usarão um segmento. Mas para programadores que escrevem programas *colossais* em linguagem assembly, o valor dessas pseudo-ops é incalculável. Se você estiver interessado, poderá encontrar mais detalhes no manual de macro assembler.

Mais para o final deste capitulo aprendemos mais sobre as raízes da instrução INT. Agora, já estamos prontos para diminuir o ritmo e aprendermos como *escrever* programas em linguagem assembly maiores e mais úteis.

12

ACERTANDO O CURSOR

Disquetes, Setores e Dskpatch

O Plano do Jogo

Resumo

Estivemos mexendo em muitas coisas novas e interessantes, por aqui e por ali, e você deve estar achando que estamos andando meio sem rumo. Mas não estamos. Já vimos o suficiente para podermos traçar um plano geral para o restante deste livro. E é isso o que faremos neste capítulo: vamos olhar de perto o projeto do nosso programa Dskpatch. Depois gastaremos o restante do livro desenvolvendo o programa, de forma semelhante ao que você fará, quando desenvolver seus próprios programas.

Não apresentaremos a versão final de Dskpatch de uma vez só; não foi assim que o escrevemos. Ao invés disso, apresentaremos pequenos programas de teste para checar cada estágio conforme prosseguimos. Para fazer isto, precisamos saber para onde vamos. Por isso, vamos corrigir o curso aqui.

Como o Dskpatch lidará com informação em disco, é por aí que vamos começar.

Disquetes, Setores e Dskpatch

A informação em seus discos está dividida em *setores*, com cada setor contendo 512 bytes de informação. Um disquete formatado dos dois lados pelo DOS a partir do 2.0 possui um total de 720 setores, ou 720 x 512 = 368.640 bytes. Se pudéssemos ver esses setores diretamente, poderíamos examinar o diretório diretamente, ou poderíamos olhar os arquivos do disco. Não podemos — não exatamente nós —, mas o Dskpatch poderá. Vamos usar o Debug para aprender mais sobre os setores e obter uma boa idéia de como iremos apresentar um setor como o Dskpatch.

O Debug possui um comando, L (*Load — Carregar*), para ler setores do disco para a memória, onde podemos olhar os dados. Como exemplo, vamos olhar o diretório de um disco de face dupla, onde se inicia no setor 5. Carregue o setor 5 do disco do drive A: (que é o drive 0 para o Debug) usando o comando L da seguinte maneira:

```
-L 100 0 5 1
```

Como você pode ver pela Figura 12-1, este comando carrega os setores para a memória, a partir do setor 5, continuando por um setor, a partir do deslocamento 100h dentro de um segmento de dados. Para apresentar na tela o setor 5, podemos usar o comando de cópia (Dump - D):

```
-D100
396F:0100 49 42 4D 42 49 4F 20 20-43 4F 4D 27 00 00 00 00 IBMBIO  COM'....
396F:0110 00 00 00 00 00 00 00 00-60 68 06 02 00 00 12 00 ........`h......
396F:0120 49 42 4D 44 4F 53 20 20-43 4F 4D 27 00 00 00 00 IBMDOS  COM'....
```

Usaremos um formato semelhante para o Dskpatch, mas com muitas melhorias. O Dskpatch será equivalente a um editor de tela completo para setores de disco. Poderemos apresentar o setor na tela e mover o cursor sobre os dados, alterando os números ou caracteres que quisermos. Também poderemos escrever este setor alterado de volta no disco, e é por isso que o chamamos de Disk Patch (Conserto em disco) — ou Dskpatch, pois o nome não pode ter mais do que oito caracteres no nome.

Dskpatch é a motivação para as rotinas que formos escrever. Ele não é de qualquer forma um fim em si. Usando Dskpatch como um exemplo para este livro, também veremos muitas rotinas que serão úteis para os seus programas. Isso significa que você verá muitas rotinas de uso geral para apresentar saídas, manipulações, entrada pelo teclado e mais.

Vamos ver mais atentamente algumas melhorias que terá a cópia de setores em relação ao Debug. A apresentação no Debug só apresenta os caracteres "imprimíveis" — 96 dos 256 caracteres diferentes que um IBM PC pode apresentar. Por que isso? Porque o MS-DOS, primo do PC-DOS, roda em muitos computadores diferentes. Alguns destes computadores apresentam apenas 96 caracteres, de modo que a Microsoft (o autor do Debug) escolheu escrever uma versão do Debug que pudesse ser executada em todas as máquinas.

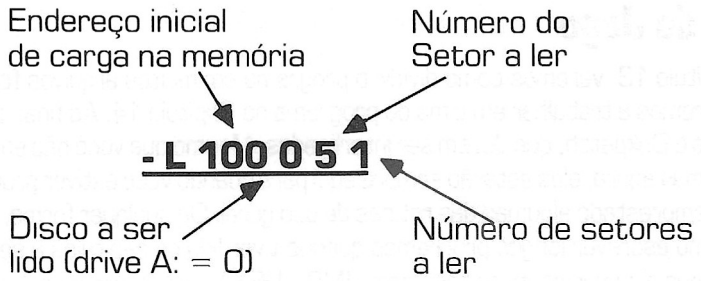


Figura 12-1. ***Comando de Carga do Debug.***

O Dskpatch e para computadores IBM PC e compatíveis, para que possamos apresentar todos os 256 caracteres isso requer um pouco de trabalho O uso da função 2 do DOS, para saída de caractere. permite-nos apresentar quase todos os caracteres, mas o DOS dá um significado especial a alguns. como o 7, que emite um bipe. Há caracteres para esses códigos especiais, e na Parte III veremos como apresenta-los.

Tambem usaremos muito as teclas de função, para que possamos, por exemplo, apresentar o próximo setor pressionando F2. E poderemos mudar qualquer byte movendo o cursor até esse byte e digitando um novo valor. Será como se estivéssemos usando um processador de texto, onde poderemos alterar os caracteres muito facilmente. A maioria desses detalhes aparecerá quando estivermos montando lentamente o programa Dskpatch. (A Figura 12-2 mostra como será uma apresentação normal do programa — uma grande melhoria em relação ao Debug.)

Disco Setor B

```
     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
00 | EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00 |
10 | 02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 |
20 | 00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 0A |
30 | D2 79 0E 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1 |
40 | 02 8E C5 8E D5 BC 00 7C 51 FC 1E 36 C5 36 78 00 |
50 | BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00 |
60 | A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1 |
70 | E0 40 E8 80 00 E8 86 00 BB 00 05 53 B0 01 E8 AB |
80 | 00 5F BE 73 01 B9 0B 00 90 F3 A6 75 62 83 C7 15 |
90 | B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B |
A0 | 00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96 |
B0 | A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 1E |
C0 | 6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52 |
D0 | F7 26 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0 |
E0 | 80 E4 03 74 04 FE C4 8A CC 5B 58 FF 2E 6F 01 BE |
F0 | 89 01 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18 |
```

Pressione tecla de função, ou entre caractere ou byte hexa:

Figura 12-2. ***Exemplo de Apresentação do Dskpatch.***

O Plano do Jogo

No Capítulo 13, veremos como dividir o programa em muitos arquivos fonte diferentes. Depois, começaremos a trabalhar em cima do programa no Capítulo 14. Ao final, teremos nove arquivos fonte para o Dskpatch, que devem ser interligados. Mesmo que você não entre e execute todos esses programas agora, eles estarão sempre aqui para quando você estiver pronto, ou quando você quiser pegar emprestado algumas das rotinas de uso geral. De qualquer forma, você terá uma idéia melhor de como escrever longos programas quando tiver lido os capítulos a seguir.

Já criamos algumas rotinas úteis, como IMP_HEXA para imprimir um byte como um número hexa de dois dígitos e IMP_DECIMAL para imprimir um número em decimal. Agora, vamos escrever alguns programas para apresentar um bloco de memória quase da mesma forma que o comando D do Debug. Começaremos apresentando 16 bytes de memória, uma linha da apresentação do Debug, e depois passaremos para 16 linhas de 16 bytes (meio setor). Um setor inteiro não cabe na tela de uma vez só no formato que escolhemos, de modo que o Dskpatch inclui rotinas para rolar a tela por um setor usando as interrupções do ROM BIOS — não as do DOS. Isso virá muito depois, após termos montado uma tela completa com meio setor.

Uma vez podendo copiar 256 bytes da memória, montaremos outra rotina para ler um setor do disco para nossa área da memória. Copiaremos metade de um setor na tela, e poderemos usar o Debug para alterar nosso programa, para que possamos copiar setores diferentes. Nesse ponto, teremos uma apresentação funcional, mas não muito atrativa, mas ela ficará melhor depois.

Com um pouco mais de trabalho e rotinas, faremos com que a tela fique muito mais agradável esteticamente. Ainda não será uma tela completa, de modo que ela também irá se perder para o alto, assim como o Debug. Mas a tela completa virá em seguida, e através dela aprenderemos sobre as rotinas do ROM BIOS, que nos permitam controlar o vídeo, mover o cursor ... esse tipo de coisas. Depois, estaremos prontos para ver como usar as rotinas do ROM BIOS a fim de imprimir todos os 256 diferentes caracteres.

A seguir virá a entrada do teclado e de comandos que nos permitirão interagir com o Dskpatch. Depois disso, precisaremos novamente corrigir o curso.

Resumo

Vimos bastante coisa do futuro aqui. Você deve agora saber melhor para onde estamos indo, de modo que vamos passar ao próximo capítulo, onde estabeleceremos a base para o projeto modular e aprenderemos a dividir um programa em muitos arquivos fonte diferentes. Depois, no Capítulo 14, escreveremos algumas rotinas de teste para apresentar seções da memória.

13

PROJETO MODULAR

Montagem Separada

As Três Leis do Projeto Modular

Resumo

Sem o projeto modular, o Dskpatch não seria muito divertido de se escrever. O uso do projeto modular facilita a tarefa de escrever qualquer programa, menos o menor. Usaremos este capítulo para definir algumas regras básicas para o projeto modular, e seguiremos essas regras pelo resto deste livro. Vamos começar aprendendo como separar um programa grande em muitos arquivos fonte diferentes.

Montagem Separada

No Capítulo 10, acrescentamos a rotina IMP_DECIMAL ao arquivo ES_VIDEO.ASM, e também uma pequena rotina chamada TESTE_IMP_DECIMAL. Vamos apanhar essa rotina de ES_VIDEO.ASM e colocá-la em um arquivo próprio, chamado TESTE.ASM. Depois, montaremos esses dois arquivos separadamente, linkeditando-os em um único programa. Eis o arquivo TESTE.ASM:

Listagem 13-1. O Arquivo TESTE.ASM

```
CODE_SEG        SEGMENT PUBLIC
        ASSUME  CS:CODE_SEG
        ORG     CS:CODE_SEG
        ORG     100h

        EXTRN   IMP_DECIMAL:NEAR

TESTE_IMP_DECIMAL       PROC    NEAR
        MOV     DX,12345
        CALL    IMP_DECIMAL
        INT     20h                     ;Retorna ao DOS
TESTE_IMP_DECIMAL       ENDP

CODE_SEG        ENDS

        END     TESTE_IMP_DECIMAL
```

Já vimos a maior parte desse arquivo fonte antes, mas uma parte dele é nova. Logo, vamos começar do alto e examinar de cima a baixo. Primeiro, a palavra *PUBLIC* aparece após SEGMENT. Isto diz ao assembler que queremos este segmento (CODE_SEG) combinado em um segmento junto com todos os outros segmentos que têm o mesmo nome — o segmento de código, neste caso. O assembler só passa essa informação ao linkeditor, que, como o seu nome indica, *linkedita* arquivos diferentes. O linkeditor realiza a junção de todas as partes diferentes.

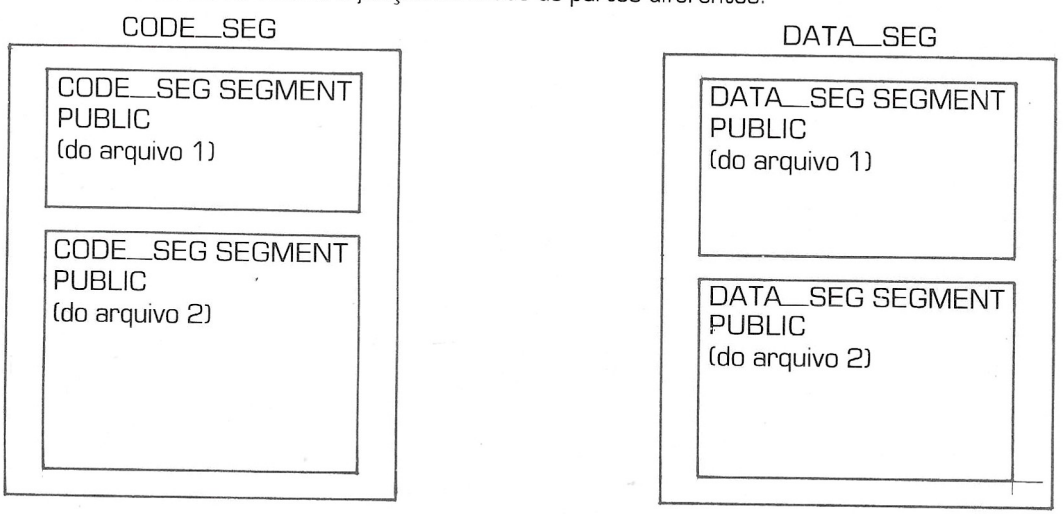


Figura 13-1. ***LINK Combina os Segmentos de Arquivos Diferentes.***

Nosso arquivo agora contém a pseudo-op EXTRN. O comando EXTRN IMP__DECIMAL:NEAR diz ao assembler duas coisas; que IMP__DECIMAL é um outro arquivo, *externo*, e que ele é definido como uma rotina NEAR nesse arquivo, de modo que deve estar no mesmo segmento. O assembler, dessa forma, gera um CALL NEAR para esta rotina; ele geraria um CALL FAR se tivéssemos colocado um FAR após IMP__DECIMAL.

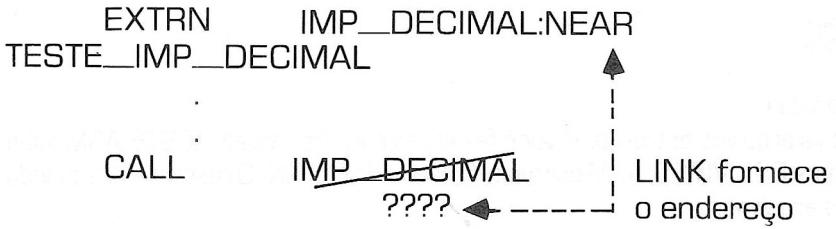


Figura 13-2. ***LINK Atribui o Endereço Para Nomes Externos.***

Estas são as únicas mudanças que precisamos fazer até que comecemos a armazenar os dados na memória. Neste ponto, apresentaremos outro segmento para os dados. Agora, vamos modificar ES__VIDEO.ASM, e depois montaremos e linkeditaremos os dois arquivos.

Remova a rotina TESTE__IMP__DECIMAL de ES__VIDEO.ASM. Já colocamos isso em TESTE.ASM, e não precisamos novamente em Es__video. Depois, remova o comando ORG 100h de Es__video. Movemos isso, também, para TESTE.ASM, que agora tem a primeira rotina do nosso programa. Como vimos no Capítulo 11, o comando ORG 100h é necessário para manter uma área de rascunho de 256 bytes em nosso programa — isto é, antes de TESTE__IMP__DECIMAL no arquivo fonte TESTE.ASM.

A seguir, temos que colocar a palavra *PUBLIC* após SEGMENT, da seguinte forma:

```
CODE__SEG       SEGMENT PUBLIC
```

de modo que o linkeditor possa saber que ele deve combinar esse segmento com o mesmo segmento em TESTE.ASM.

Finalmente, mudamos END TESTE__IMP__DECIMAL no final de ES__VIDEO.ASM para somente END. Mais uma vez, movemos esta rotina principal para TESTE.ASM. As rotinas em ES__VIDEO são agora rotinas *externas*, e nada mais. Isto é, elas não possuem função própria; elas devem ser ligadas às rotinas que as chamam de outros arquivos. Não precisamos de um nome após a pseudo-op END em ES__VIDEO.ASM, pois nosso programa principal agora é TESTE.ASM.

Quando você acabar de fazer estas mudanças, seu arquivo fonte ES__VIDEO.ASM deve estar mais ou menos assim:

```
CODE__SEG       SEGMENT PUBLIC
        ASSUME  CS:CODE__SEG

        PUBLIC  IMP__DIG__HEXA
          .
          .
IMP__DIG__HEXA      ENDP
        PUBLIC  IMP__HEXA
          .
          .
IMP__HEXA       ENDP
```

```
        PUBLIC  IMP_CAR

IMP_CAR         ENDP
        PUBLIC  IMP_DECIMAL

IMP_DECIMAL     ENDP
CODE_SEG        ENDS
        END
```

com um ASSUME no início.

Monte estes dois arquivos assim como você fez antes para Es_video. TESTE.ASM sabe tudo o que precisa sobre ES_VIDEO.ASM por meio do comando EXTRN. O restante virá quando linkeditarmos os dois arquivos.

Agora você deve ter os arquivos TESTE.OBJ e ES_VIDEO.OBJ. Use o comando a seguir para linkeditar os dois arquivos, gerando um programa chamado TESTE.EXE.

```
A>LINK TESTE ES_VIDEO;
```

LINK junta as rotinas desses dois arquivos para criar um arquivo contendo o programa inteiro. Ele usa o nome do primeiro arquivo que entramos com o nome do arquivo .EXE resultante, de modo que temos agora TESTE.EXE.

Finalmente, crie um arquivo .COM, como você havia feito, digitando *EXE2BIN TESTE TESTE.COM*. E é isso, criamos um programa a partir de dois arquivos fonte. O programa .COM final é idêntico à versão que criamos a partir do único arquivo ES_VIDEO.ASM, quando ele tinha a rotina principal TESTE_IMP_DECIMAL.

Daqui para frente, usaremos muito os arquivos fonte separados, e seu valor se tornará mais claro quando as rotinas se "amontoarem". No próximo capítulo, escreveremos um programa de teste para copiar seções da memória em hexa. Geralmente, escreveremos uma versão simples de teste sempre antes de escrevermos a versão completa. Isso nos permite ver como escrever uma boa versão final, bem como economizar muito esforço e perturbação mental no processo.

Há diversas outras maneiras de se diminuir o esforço. Nós as chamamos de *As Três Leis do Projeto Modular*.

As Três Leis do Projeto Modular

Essas leis são resumidas na Tabela 13-1. Elas não são realmente *leis*, mas sugestões. Iremos, porém, usá-las neste livro como se fossem realmente leis. Defina suas próprias leis, se quiser, mas de qualquer forma, seja fiel às mesmas o tempo inteiro. Sua tarefa será muito mais fácil se você for coerente.

Há um paralelo óbvio entre o projeto modular na programação e o projeto modular na engenharia. Um engenheiro elétrico, por exemplo, pode montar uma peça muito complicada de um equipamento com caixas que executam diferentes funções, sem saber como funciona exatamente cada uma delas. Mas se cada caixa usar voltagens e conexões diferentes, a falta de coerência cria um motivo para dor de cabeça ao pobre engenheiro, que deve, de algum modo, fornecer voltagens diferentes para cada caixa, ou criar conexões especiais entre as caixas (ou circuitos). Não é muito divertido, mas, felizmente para o engenheiro, há padrões de voltagem fixando valores para os circuitos. Assim, somente algumas poucas voltagens devem ser necessárias, ao invés de uma voltagem diferente para cada circuito (ou caixa).

Tabela 13-1. As Três Leis do Projeto Modular

1. Salve e recupere todos os registradores, a menos que a rotina retorne um valor por meio desse registrador.
2. Seja coerente com relação aos registradores que você usa para passar informação. Por exemplo:
   — DL,DX — Envia valores de byte e palavra.
   — AL,AX — Retorna valores de byte e palavra.
   — BX:AX — Retorna valores de dupla palavra.
   — DS:DX — Envia e retorna endereços.
   — CX — Mantém valores de contadores em geral.
   — CF — Setado quando há um erro; um código de erro deve ser retornado em um dos registradores, como AL ou AX.
3. Defina todas as interações externas no cabeçalho do comando:
   — Informação necessária na entrada.
   — Informação retornada (registradores alterados).
   — Rotinas chamadas.
   — Variáveis usadas (lidas, escritas etc.).

O projeto modular e as interfaces-padrão são igualmente importantes em programas em linguagem assembly, e é por isso que estabelecemos as leis (por assim dizer) e usaremos essas leis daqui por diante. Como você entenderá ao final do livro, essas regras tornarão nossa tarefa muito mais simples. Vamos dar uma olhada nessas leis com mais detalhe.

Salve e recupere *todos* os registradores, a menos que a rotina retorne um valor nesse registrador. Não há muitos registradores no 8088. Salvando os registradores no início de uma rotina nós os deixamos livres para serem usados à vontade. Mas devemos ter o cuidado de restaurá-los ao final da rotina. Você poderá ver isso em todas as nossas rotinas, com instruções PUSH no início da rotina e POPs no final.

A única exceção é para rotinas que devem retornar alguma informação à rotina que a chama. Por exemplo, uma rotina que lê um caractere do teclado deve retornar o caractere de algum modo. Não iremos salvar registradores usados para retornar alguma informação.

Rotinas curtas também evitam o problema de falta de registradores. Às vezes, usaremos uma rotina que é usada apenas uma vez. Isto não somente ajuda a lidar com a falta de registradores, como também torna o programa mais fácil de escrever e entender. Veremos mais sobre isso quando escrevermos as rotinas para Dskpatch.

Seja coerente com relação aos registradores que você usa para passar informação. Nossa tarefa torna-se mais simples se definirmos padrões para a troca de informação entre as rotinas. Usaremos um registrador para enviar informação, e um para receber informação. Também precisaremos enviar endereços para grandes porções de dados, e para isso usaremos o par de registradores DS:DX, para que os nossos dados possam estar em qualquer lugar da memória. Você verá mais sobre isso quando apresentarmos um novo segmento para os dados e começarmos a usar o registrador DS.

Geralmente reservamos o registrador CX para contagens de repetição. Logo escreveremos uma rotina para imprimir um caractere diversas vezes, de modo que podemos imprimir dez espaços chamando essa rotina (IMP_CAR_N_VEZES) com CX setado em 10. Usaremos o registrador CX sempre que tivermos que repetir uma contagem ou quando quisermos retornar algum contador,

como o número de caracteres lidos do teclado (faremos isso quando escrevermos uma rotina chamada LE_STRING).

Finalmente, setaremos o Flag de Carry (CF), sempre que houver um erro, e o zeraremos quando não houver um erro. Nem todas as rotinas usam os flags de carry. Por exemplo, IMP_CAR sempre funciona, e não há motivo para retornar um sinal de erro. Mas uma rotina que grava algo no disco pode encontrar muitos erros (nenhum disco, proteção de escrita etc.). Neste caso, usaremos um registrador para retornar um código de erro. Não há um padrão aqui, pois o DOS usa registradores diferentes para funções diferentes. É um erro dele, não nosso.

Defina todas as interações externas no cabeçalho do comando. Não há motivo para sabermos como uma rotina funciona se tudo o que queremos é usá-la, e é por isso que colocamos um cabeçalho de comentário detalhado antes de cada rotina. Este cabeçalho contém toda a informação de que precisamos saber. Ele nos diz o que colocar em cada registrador antes de chamar a rotina, e diz que informação a rotina retorna. A maioria das rotinas usa registradores como suas variáveis, mas algumas usam variáveis na memória. O comentário no cabeçalho deve informar quais dessas variáveis na memória são lidas e quais são alteradas. E por último, cada lista de cabeçalho deve apresentar as outras rotinas chamadas. Eis um exemplo de cabeçalho com as informações completas:

```
;-----------------------------------------------------------------------;
; Este é um exemplo de cabeçalho completo. Esta parte normalmente seria uma breve descrição do que a rotina faz. Por ;
; exemplo, esta rotina imprimiria "Setor " na primeira linha.           ;
;                                                                       ;
;                                   DS:DX   Endereço da mensagem "Setor " ;
;                                                                       ;
; Chama:        VAL_XY, IMP_STRING (rotinas chamadas                    ;
; Le:           LINHA_STATUS       (variaveis lidas memoria)            ;
; Escreve:      NULO               (variaveis alteradas na memoria)     ;
;                                                                       ;
;-----------------------------------------------------------------------;
```

Sempre que quisermos usar qualquer rotina escrita, podemos olhar para esse comentário e saber tudo do que precisamos. Não há necessidade de analisarmos o funcionamento interno da rotina para ver como ela funciona.

Essas leis tornam a programação em linguagem assembly mais fácil, e certamente iremos cumpri-las, mas não necessariamente na primeira tentativa. A primeira versão de uma rotina ou programa é um teste. Em geral, não saberemos como escrever exatamente o programa que temos em mente, e um "esboço grosseiro" não terá que se comprometer com as leis do projeto modular. Vamos simplesmente trabalhar no programa e tentar obter algo que funcione. Depois, podemos voltar e escrever cada rotina segundo essas leis.

A programação é um processo que passa rapidamente. No decorrer deste lviro, mostraremos muitos dos processos usados em Dskpatch, mas certamente não poderemos mostrar tudo. Não há espaço suficiente para mostrarmos todas as versões escritas antes do surgimento da versão final. Via de regra, nossa primeira tentativa parece-se muito pouco com a versão final que você verá, de modo que quando estiver fazendo seus próprios programas não se preocupe em fazer tudo correto da primeira vez. Esteja preparado para reescrever cada rotina conforme entende melhor o que deseja fazer.

No próximo capítulo, montaremos um programa de teste simples para imprimir um bloco da memória. Não será a versão final; passaremos por outras até estarmos satisfeitos, e mesmo assim, poderá haver mudanças a fazer. A moral é: Um programa nunca está acabado... mas temos que parar em algum ponto.

Resumo

Este foi um capítulo para você se lembrar e usar no futuro. Começamos aprendendo como separar um programa em diversos arquivos fonte diferentes, que podem ser montados separadamente, e depois unidos por meio do linkeditor. Usamos as pseudo-ops PUBLIC e EXTRN para informar ao linkeditor que há conexões entre diferentes arquivos fonte. PUBLIC diz que outros arquivos fonte podem emitir um CALL para as rotinas indicadas após os PUBLICs, enquanto EXTRN diz que a rotina que queremos usar está em outro arquivo.

Também usamos PUBLIC após a definição SEGMENT para que o linkeditor junte os segmentos do mesmo nome que estão em arquivos diferentes.

Depois passamos para as três Leis do Projeto Modular. Estas regras foram feitas para simplificar a tarefa da programação, de forma que você deverá usá-las em seus programas assim como estão sendo usadas aqui. Você verá que é mais fácil escrever, depurar e ler programas se eles forem adaptados a essas Três Leis.

14

COPIANDO A MEMÓRIA

Modos de Endereçamento

Adicionando Caracteres à Cópia

Copiando 256 Bytes da Memória

Resumo

Daqui para a frente, iremos nos concentrar na montagem do programa Dskpatch, de certa forma do mesmo modo como fizemos com o original escrito. Algumas instruções nas rotinas a seguir podem não ser familiares, mas voltaremos a elas quando estudarmos o programa, mas para uma explicação detalhada, você precisará de um livro que cubra todas as instruções em detalhes.

Ao invés de vermos todas as instruções do 8088, vamos nos concentrar nos novos conceitos, como os diferentes modos de se endereçar a memória, que veremos neste capítulo. Na Parte III, iremos ainda além do nível das instruções, e veremos detalhes que se aplicam ao IBM PC e seus compatíveis mais próximos.

Agora, vamos aprender sobre os *modos de endereçamento*, escrevendo um pequeno programa para copiar 16 bytes de memória em notação hexa. Para começar, precisamos aprender a usar a memória como variáveis.

Modos de Endereçamento

Já vimos dois modos de endereçamento; eles são conhecidos como modo de endereçamento de *registrador* e *imediato*. O primeiro que aprendemos foi o modo registrador, que usa registradores como variáveis. Por exemplo, a instrução:

```
MOV     AX,BX
```

usa dois registradores, AX e BX, como variáveis.

Depois, passamos para o modo imediato, em que movemos um número para um registrador, conforme o exemplo:

```
MOV     AX,2
```

Isso move um byte ou palavra de memória *imediatamente* após a instrução para o registrador. Desta forma, a instrução MOV acima possui um byte, e os outros dois bytes correspondem à representação do dado imediato (0002):

```
396F:0100 B80200        MOV     AX,0002
```

A instrução é B8h, e os dois bytes seguintes são 02h e 00h (lembre-se de que o 8088 armazena os bytes em ordem inversa ao que escrevemos, com o byte de baixa ordem em primeiro lugar).

Agora, vejamos como usar a memória como uma variável. O modo imediato nos permite ler uma parte fixa da memória imediatamente após a instrução, mas não nos permite alterar a memória. Para isso, precisaremos de outros modos de endereçamento.

Vamos começar com um exemplo. O programa a seguir lê 16 bytes de memória, um de cada vez, e apresenta cada byte em notação hexa, deixando um espaço entre os 16 bytes. Entre com o programa no arquivo DISP_SET.ASM e use o assembler para montá-lo. Depois, vamos precisar mexer mais um pouco em ES_VIDEO, mas por enquanto, vamos nos preocupar somente com DISP_SET.ASM:

Listagem 14-1. O Novo Arquivo DISP_SET.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG      ;Agrupa dois segmentos
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT     PUBLIC
        ORG     100h

        EXTRN   IMP_HEXA:NEAR
        EXTRN   IMP_CAR:NEAR

;-----------------------------------------------------------------
;  Este é um programa de teste para copiar 16 bytes de memória em
;  forma hexa, em uma linha.
;-----------------------------------------------------------------
```

Listagem 14-1. *Continuação*

```
DISP_LIN        PROC    NEAR
        XOR     BX,BX                   ;Zera BX
        MOV     CX,16                   ;Copia 16 bytes
LOOP_HEXA
        MOV     DL,SETOR[BX]            ;Pega 1 byte
        CALL    IMP_HEXA                ;Copia o byte em hexa
        MOV     DL,' '                  ;Espaço entre os numeros
        CALL    IMP_CAR
        INC     BX
        LOOP    LOOP_HEXA
        INT     20h                     ;Retorna ao DOS
DISP_LIN        ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
        PUBLIC  SETOR
SETOR   DB      10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h  ;Valores do teste
        DB      18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
DATA_SEG        ENDS

        END     DISP_LINE
```

Note que colocamos o segmento de dados (DATA_SEG) *após* o segmento de código (CODE_SEG). Nós o colocamos ao final do arquivo para que o linkeditor carregue os dados na memória ao final do nosso programa.

Acrescentamos alguns novos truques a este programa, e portanto precisamos alterar um pouco o programa ES_VIDEO.ASM. Primeiro, remova o comando ASSUME de ES_VIDEO e coloque estas duas linhas no início:

```
CGROUP      GROUP     CODE_SEG      ; Agrupa dois segmentos
            ASSUME    CS:CGROUP
```

Colocaremos estas duas linhas no começo de cada arquivo daqui para a frente, com uma pequena variação. Iremos escrever:

```
CGROUP      GROUP     CODE_SEG, DATA SEG      ; Agrupa dois segmentos
            ASSUME    DS:CGROUP
```

(com DATA_SEG) sempre que tivermos o segmento de código e de dados no arquivo.

O ASSUME aqui está no lugar do antigo, e veremos depois o que estes dois comandos realmente fazem. Mas, agora, vamos testar nosso novo programa para ver como ele funciona. Use o assembler para Es_video e Disp_set.

Estamos prontos para linkeditar ES_VIDEO.OBJ e DISP_SET.OBJ e executar o resultado através de Exe2bin. Assim, use o LINK para criar um arquivo .EXE chamado DISP_SET.EXE. O primeiro nome de arquivo no comando LINK deve ser o nome do arquivo que contém a rotina principal (Disp_set nesse caso), e o ponto-e-vírgula deve aparecer ao final da linha; portanto digite:

```
A>LINK DISP_SET ES_VIDEO;
```

A linkedição sempre será a mesma, com mais nomes de arquivo antes do ponto-e-vírgula quando houver mais arquivos, mas a rotina principal deve sempre ser a primeira na lista.

Agora, converta o arquivo .EXE para um arquivo .COM digitando:

```
A>EXE2BIN DISP_SET DISP_SET.COM
```

Em geral, os dois passos anteriores para qualquer quantidade de arquivos serão:

```
LINK arq1 arq2 arq3 ...;
EXE2BIN arq1 arq1.COM
```

Agora, execute o arquivo .COM, tendo *antes* executado Exe2hin Disp_set. Caso contrário, você estará executando a versão .EXE do programa, e ninguém sabe o que poderá acontecer. O pior

que pode acontecer é você ter que desligar o computador, esperar um minuto, e depois ligá-lo novamente.

Se você não vir:

```
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
```

quando executar o programa, volte e verifique cuidadosamente qual foi o erro cometido.

Agora, vejamos como funciona o programa Disp_set. A instrução:

```
MOV     DL,SETOR[BX]    ;Pega 1 byte
```

usa um novo modo de endereçamento chamado *Endereçamento Indireto de Memória* — endereçando a memória por meio do registrador de *Base* com *deslocamento* ou, de forma mais simples, *Relativo à Base*. Vejamos o que isso realmente significa.

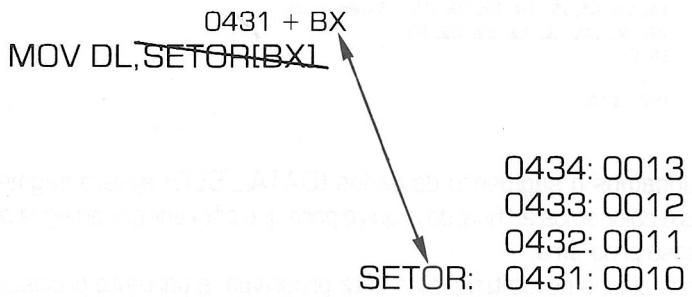


Figura 14-1. ***Tradução de SETOR[BX]***

Olhando em Disp_set, você verá queo rótulo SETOR está em um segmento chamado DATA_SEG. Este é um novo segmento usado para variáveis na memória. Toda vez que quisermos colocar e ler dados da memória, deixaremos algum espaço nesse segmento. Voltaremos às variáveis na memória logo em seguida, mas antes vamos ver algo mais sobre os segmentos.

O ASSUME DS:CGROUP diz ao assembler onde encontrar as variáveis da memória. Você deve ter pensado que iríamos querer usar ASSUME DS:DATA_SEG, mas não. Por estarmos montando um arquivo .COM, só devemos usar um segmento. Mas é conveniente trabalhar com dois: um para o código e outro para os dados. E é aí que o pseudo-op GROUP entra em ação. GROUP agrupa diferentes segmentos no que é efetivamente um segmento só, com o nome que damos antes de GROUP. Assim, o comando:

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
```

combina os dois segmentos indicados em um único segmento real de 64K com o nome CGROUP. O funcionamento interno dos grupos é um pouco mais complicado, mas não entraremos em maiores detalhes. Se você quiser os detalhes, poderá encontrá-los no seu manual de macro assembler. Cuidado, no entanto: eles são um tanto difíceis de se ler.

É hora de voltarmos ao nosso modo de endereçamento relativo à base. As duas linhas:

```
SETOR   DB   10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h   ;Valores de teste
        DB   18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
```

reservam 16 bytes de dados na memória a partir de SETOR, que o assembler converte em um espaço. DB, como você deve se lembrar, significa *Define Byte*; os números após cada DB são valores iniciais. Assim, quando você inicia o programa, a parte da memória começando em SETOR irá conter 10h, 11h, 12h, e assim por diante. Se tivéssemos escrito:

```
MOV     DL,SETOR
```

a instrução moveria o primeiro byte (10h) para o registrador DL. Isso é conhecido como modo de endereçamento *direto*. Mas não fizemos assim. Colocamos [BX] após a palavra SETOR. Isso torna o comando parecido com um vetor indexado, como em programas BASIC:

```
DL = SETOR(10)
```

que move o décimo elemento de SETOR em DL.

De fato, nossa instrução MOV é muito parecida. O registrador BX contém um *deslocamento* na memória com relação a SETOR. Assim, se BX for 0, o MOV DL,SETOR[BX] move o primeiro byte (10h aqui) para DL. Se BX for 0Ah, esta instrução MOV moverá o décimo primeiro elemento (1Ah — lembre-se, começamos com 0) para DL.

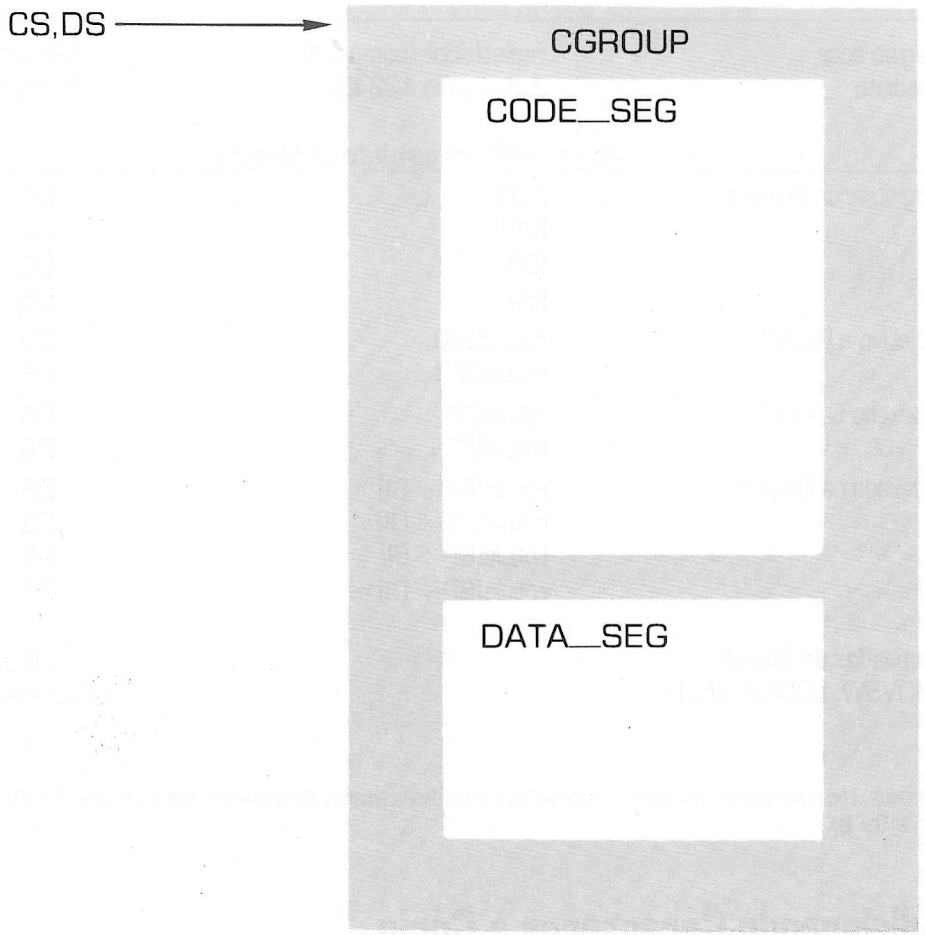


Figura 14-2. ***GROUPS Trata Múltiplos Segmentos Como um Único Segmento.***

Por outro lado, a instrução MOV DX,SETOR[BX] moveria a sexta palavra para DX, pois um deslocamento de 10 bytes é o mesmo que cinco palavras, e a primeira palavra está no deslocamento zero. (Para os entusiastas: Esta última instrução MOV não é válida, pois SETOR é um rótulo de

byte, enquanto DX é um registrador de palavras. Teríamos que escrever MOV DX,Word Ptr SETOR[BX] para dizer ao assembler que realmente queremos usar setor como um rótulo de palavra nesta instrução, e não byte.)

Há muitos outros modos de endereçamento; alguns deles nós veremos depois, mas a maioria não. Todos os modos de endereçamento estão listados na Tabela 14-1.

Tabela 14-1. Modos de Endereçamento

Modo de Endereçamento	Formato do Endereço	Registrador de Segmento Usado
Registrador	registrador (como AX)	Nenhum
Imediato	dado (como 12345)	Nenhum
Modos de Endereçamento da Memória		
Registrador Indireto	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Relativo a Base *	rótulo[BX]	DS
	rótulo[BP]	SS
Indexado Direto *	rótulo[DI]	DS
	rótulo[SI]	DS
Indexado na Base *	rótulo[BX + SI]	DS
	rótulo[BX + DI]	DS
	rótulo[BP + SI]	SS
	rótulo[BP + DI]	SS
Comandos de String: (MOVSW, LODSB, *etc.*)		Lidos de DS:SI Escritos em ES:DI

* Rótulo[...] pode ser trocado por [desl + ...], onde "desl" é um deslocamento. Assim poderíamos escrever [10 + BX] e o endereço seria 10 + BX.

Adicionando Caracteres à Cópia

Estamos quase terminando a rotina para uma cópia da memória conforme o Debug. Até aqui, copiamos os dígitos hexa em uma linha; no próximo passo colocaremos os caracteres após os dígitos hexa. Portanto, vamos passar logo à nova versão DISP_LIN (em DISP_SET.ASM) com um segundo loop acrescentado, para a apresentação dos caracteres:

Listagem 14.2. Mudanças na Rotina DISP_LIN de DISP_SET.ASM

```
DISP_LIN        PROC    NEAR
        XOR     DX,BX           ;Zera BX
        MOV     CX,16           ;Copia 16 bytes
```

```
LOOP_HEXA:
        MOV     DL,SETOR[BX]    ;Pega 1 byte
        CALL    IMP_HEXA        ;Copia o byte em hexa
        MOV     DL,' '          ;Espaco entre os numeros
        CALL    IMP_CAR
        INC     BX
        LOOP    LOOP_HEXA

        MOV     BL,' '          ;Acrescenta espaco antes dos caracteres
        CALL    IMP_CAR
        MOV     CX,16
        XOR     BX,BX           ;Zera BX
LOOP_ASCII:
        MOV     DL,SETOR[BX]
        CALL    IMP_CAR
        INC     BX
        LOOP    LOOP_ASCII
        INT     20h             ;Retorna ao DOS
DISP_LIN        ENDP
```

Monte o programa, linkedite-o com Es_video, execute-o através de Exe2bin, e faça um teste. Justamente a tela que queríamos. (Veja a Figura 14-3.)

```
A>disp_set
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
A>
```

Figura 14-3. ***Saída de DISP_LIN.***

Tente alterar os dados para incluir 0Dh ou 0Ah, você verá alguns caracteres estranhos. Eis por que: 0Dh e 0Ah são os caracteres para mudança-de-linha e retorno-de-carro. O DOS interpreta esses caracteres como comandos para mover o cursor, mas aqui só queremos ver os caracteres correspondentes. Para isso, temos que alterar IMP_CAR para que imprima todos os caracteres, sem aplicar qualquer significado especial. Faremos isso na Parte III, mas por enquanto vamos reescrever IMP_CAR de forma um pouco diferente, para que ele imprima um ponto no lugar dos caracteres baixos (entre 0 e 1Fh).

Troque a rotina IMP_CAR em ES_VIDEO.ASM por esta nova rotina:

Listagem 14-3. Uma Nova Rotina IMP_CAR em ES_VIDEO.ASM

```
        PUBLIC  IMP_CAR

;-------------------------------------------------------------------------
; Esta rotina imprime um caractere na tela usando
; a chamada de função do DOS. IMP_CAR troca os caracteres
; de 0 a 1Fh por um ponto.
;       DL      byte impresso na tela.
;-------------------------------------------------------------------------
IMP_CAR PROC    NEAR
        PUSH    AX
        PUSH    DX
        CMP     DL,32           ;O caractere esta antes do espaco?
        JAE     IMPRESSO        ;Nao, imprime.
        MOV     DL,'.'          ;Sim, troca por ponto.
IMPRESSO:
        MOV     AH,2            ;Chamada para saida de caractere
        INT     21h             ;Sai caractere em DL
        POP     DX              ;Recupera valores de AX e DX
        POP     AX
        RET
IMP_CAR ENDP
```

Teste essa nova rotina com Disp_set, e altere os dados para vários caracteres e verifique os limites de condição.

```
A>disp_set
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
A>
```

Figura 14-4. ***Versão Modificada de DISP_LIN.***

Copiando 256 Bytes da Memória

Até aqui, só trabalhamos na cópia de uma linha, ou 16 bytes da memória. O próximo passo será copiar 256 bytes de memória, que é exatamente metade de um setor. Portanto vamos trabalhar no sentido de copiar metade de um setor. Ainda temos muitas melhorias a fazer; esta é só uma versão de teste.

Precisaremos de duas novas rotinas aqui, e uma versão modificada de DISP_LIN. As novas rotinas são DISP_MEIO_SETOR, que logo se tornará uma rotina pronta para apresentar meio setor, e ENVIA_CRLF, que muda o cursor para o início da linha seguinte (CRLF significa *Carriage Return-Line Feed* — ou seja, *Retorno de carro e Mudança de linha*, os caracteres que movem o cursor para a linha seguinte).[1]

ENVIA_CRLF é muito simples, e por isso vamos começar com ele. Coloque a seguinte rotina em um arquivo chamado CURSOR.ASM:

Listagem 14-4. O Novo Arquivo CURSOR.ASM

```
CR              EQU     13      ;Retorno de carro
LF              EQU     10      ;Mudança de linha
CGROUP  GROUP   CODE_SEG
        ASSUME  CS:CGROUP

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  ENVIA_CRLF

;-------------------------------------------------------------------------
; Esta rotina envia um par CR/LF para o video,                           ;
; usando as rotinas do DOS para que a tela seja                          ;
; passada corretamente.                                                  ;
;-------------------------------------------------------------------------
ENVIA_CRLF      PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     AH,2
        MOV     DL,CR
        INT     21h
        MOV     DL,LF
        INT     21h
        POP     DX
        POP     AX
        RET
ENVIA_CRLF      ENDP

CODE_SEG        ENDS

        END
```

[1] N.T. Os termos CR e LF não são traduzidos aqui, juntamente com outros termos (CS, DS, ES, SS, SP, BP etc.) no decorrer deste livro. O motivo é que tais termos tornaram-se comuns à terminologia em computadores. Além disso, alguns deles podem causar confusão, se traduzidos; por exemplo, se Segmento de Pilha for representado por SP, este termo pode ser entendido como "Stack Pointer" (tradicionalmente, SP).

Esta rotina envia um par de caracteres de Retorno de Carro e Mudança de Linha usando a função 2 do DOS para enviar caracteres. O comando:

```
CR              EQU   13 10     ;Retorno de Carro
```

usa a pseudo-op EQU para definir um nome CR como sendo igual a 13. Assim, a instrução MOV DL,CR é a mesma que MOV DL,13. Como a Figura 14-5 nos mostra, o assembler substitui por um 13 sempre que encontrar CR. Da mesma forma, sempre que ele encontrar um LF, ele fará a troca por 10.

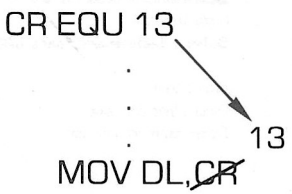


Figura 14-5. ***A Pseudo-op EQU nos Permite Usar Nomes no Lugar de Números.***

O arquivo Disp_set precisa agora de muito trabalho. Eis a nova versão de DISP_SET.ASM. Daqui em diante, os acréscimos ao programa serão indicados com texto em negrito; o texto que deve ser deletado será impresso em itálico:

Listagem 14-5. A Nova Versão de DISP_SET.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG  ;Agrupa dois segmentos
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        ORG     100h

        PUBLIC  DISP_MEIO_SETOR
        EXTRN   ENVIA_CRLF:NEAR

;-------------------------------------------------------------------------------
; Esta rotina apresenta meio setor (256 bytes)                                 ;
;                                                                              ;
; Usa:          DISP_LIN, ENVIA_CRLF                                           ;
;-------------------------------------------------------------------------------
DISP_MEIO_SETOR
                PROC    NEAR
        XOR     DX,DX           ;Começa no início de SETOR
        MOV     CX,16           ;Mostra 16 linhas
MEIO_SETOR:
        CALL    DISP_LIN
        CALL    ENVIA_CRLF
        ADD     DX,16
        LOOP    MEIO_SETOR
        INT     20h ENDP
DISP_MEIO_SETOR
        PUBLIC  DISP_LIN
        EXTRN   IMP_HEXA:NEAR
        EXTRN   IMP_CAR:NEAR
```

```
;-----------------------------------------------------------------------------
; Esta rotina mostra uma linha de dados, ou 16 bytes, primeiro
; em hexa, e depois em ASCII
;
;       DS:DX     Deslocamento no setor, em bytes.
;
; Usa:            IMP_CAR, IMP_HEXA
; Le              SETOR
;-----------------------------------------------------------------------------
DISP_LIN        PROC    NEAR
        XOR     BX,DX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX           ;Deslocamento mais util em BX
        MOV     CX,16           ;Copia 16 bytes
        PUSH    BX              ;Salva o deslocamento para LOOP_ASCII
LOOP_HEXA:
        MOV     DL,SETOR[BX]    ;Pega 1 byte
        CALL    IMP_HEXA        ;Copia o byte em hexa
        MOV     DL,' '          ;Espaco entre os numeros
        CALL    IMP_CAR
        INC     BX
        LOOP    LOOP_HEXA

        MOV     DL,' '          ;Acrescenta espaco antes dos caracteres
        CALL    IMP_CAR
        MOV     CX,16
        POP     BX              ;Volta com o deslocamento no SETOR
        XOR     BX,BX
LOOP_ASCII:
        MOV     DL,SETOR[BX]
        CALL    IMP_CAR
        BX      LOOP_ASCII

        POP     DX
        POP     CX
        POP     BX
        RET
        INT     20h
DIP_LIN
                ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
        PUBLIC  SETOR
SETOR   DB      16h, 19h, 12h, 13h, 14h, 15h, 16h, 17h  ; Test pattern

SETOR   DB      16 DUP(10h)
        DB      16 DUP(11h)
        DB      16 DUP(12h)
        DB      16 DUP(13h)
        DB      16 DUP(14h)
        DB      16 DUP(15h)
        DB      16 DUP(16h)
        DB      16 DUP(17h)
        DB      16 DUP(18h)
        DB      16 DUP(19h)
        DB      16 DUP(1Ah)
        DB      16 DUP(1Bh)
        DB      16 DUP(1Ch)
        DB      16 DUP(1Dh)
        DB      16 DUP(1Eh)
        DB      16 DUP(1Fh)
DATA_SEG        ENDS
        END     DISP_MEIO_SETOR
```

As alterações aqui são bem simples. Em DISP_LIN, acrescentamos um PUSH BX e um POP BX em torno de LOOP_HEXA, pois queremos reutilizar o deslocamento inicial em LOOP_ASCII. Também acrescentamos instruções PUSH e POP para salvar e recuperar todos os registradores que usamos dentro de DISP_LIN. Na verdade, DISP_LIN está quase pronta; as únicas mudanças a fazer são estéticas, para adicionar espaços e caracteres gráficos para que vejamos uma apresentação mais atraente; isso virá em breve.

Quando você linkeditar os arquivos, lembre-se que agora temos três arquivos: Disp_set, Es_video e Cursor. Disp_set deve ser o primeiro na lista. Após você executar a versão .EXE por meio de Exe2bin, você verá uma tela como a da Figura 14-6.

Teremos mais arquivos antes de terminarmos, mas agora vamos passar ao próximo capítulo, onde copiaremos um setor diretamente do disco para a memória, antes de apresentarmos meio setor na tela.

```
A> disp_set
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10  ................
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11  ................
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12  ................
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13  ................
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14  ................
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15  ................
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16  ................
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17  ................
18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18  ................
19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19  ................
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A  ................
1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B  ................
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C  ................
1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D  ................
1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E  ................
1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F  ................

A>
```

Figura 14-6. ***Saída de Disp_set.***

Resumo

Já sabemos mais sobre os modos de endereçamento para uso da memória e dos registradores do 8088. Aprendemos sobre o modo de endereçamento indireto, que usamos inicialmente para ler 16 bytes da memória.

Também usamos o modo de endereçamento indireto em diversos programas deste capítulo, começando com o programa para imprimir 16 dígitos hexa na tela. Estes 16 números vieram de uma área da memória rotulada como SETOR, que expandimos um pouco depois para que apresentasse 256 bytes — meio setor.

E, por último, começamos a ver cópias na tela, conforme elas aparecem no seu vídeo, ao invés de textos impressos em tipos. Usaremos essas cópias da tela nos capítulos seguintes, para que possamos aproveitar melhor o resultado final.

15

COPIANDO UM SETOR DO DISCO

Tornando a Vida Mais Fácil

Formato do Arquivo Make

Alterando o Disp_set

Lendo um Setor

Resumo

Agora que já temos um programa que copia 256 bytes de memória, podemos adicionar algumas rotinas para ler um setor do disco e colocá-lo na memória a partir do início de SETOR. Depois, nossas rotinas de cópia encarregar-se-ão de apresentar a primeira metade deste setor.

Tornando a Vida Mais Fácil

Com os três arquivos que tínhamos no capítulo anterior, a vida tornou-se um pouco difícil. Nós alteramos os três arquivos, ou somente dois? Você provavelmente "assemblou" todos os três, ao invés de checar para ver se houve mudanças desde a última montagem.

Mas se formos montar todos os arquivos sempre que fizermos uma modificação em um somente, o processo será muito lento, e tornar-se-á ainda mais lento, conforme o Dskpatch se torna ainda maior. O que realmente queremos é montar apenas os arquivos que tenham sido alterados.

Felizmente, se você estiver usando um dos mais recentes pacotes de Macro Assembler da Microsoft (ou possui o Compilador C da Microsoft), há um modo de fazer exatamente isso. Eles incluem um programa chamado Make, que faz exatamente o que queremos. Para usá-lo, criamos um arquivo (que chamaremos de Dskpatch) que diz ao Make como fazer o seu trabalho, e depois simplesmente digitamos:

```
A>MAKE DSKPATCH
```

Make, em seguida, só irá montar os arquivos que foram alterados.

O arquivo que você cria (Dskpatch) diz ao Make quais os arquivos que dependem de outros arquivos. Toda vez que você modifica um arquivo, o DOS altera a hora em que você fez a atualização (você pode ver isso na listagem do DIR). Make simplesmente olha para as versões .ASM e .OBJ de um arquivo. Se a versão .ASM possui uma hora de modificação mais recente que a versão .OBJ, ele sabe que precisa montar esse arquivo novamente.

E isso é tudo, mas há um aviso que precisa ser feito: Make só funcionará corretamente se você tiver o cuidado de manter sempre atualizadas a data e a hora do sistema cada vez que inicializar o computador (ligando-o ou resetando-o). Sem essa informação, o Make nem sempre saberá quando você fez mudanças em um determinado arquivo.

Formato do Arquivo Make

O formato do nosso arquivo, Dskpatch, que usaremos com o Make, é muito simples:

Listagem 15-1. O Arquivo Make DSKPATCH

```
disp_set.obj:       disp_set.asm
        masm disp_set;

es_video.obj:       es_video.asm
        masm es_video;

cursor.obj:         cursor.asm
        masm cursor;

disp_set.com:       disp_set.obj es_video.obj cursor.obj
        link disp_set es_video cursor;
        exe2bin disp_set disp_set.com
```

Cada entrada possui um nome de arquivo à esquerda (antes dos dois pontos) e um ou mais nomes de arquivo à direita. Se qualquer um dos arquivos à direita (como DISP_SET.ASM na primeira linha) for mais recente do que o primeiro arquivo (DISP_SET.OBJ), Make executará todos os comandos endentados que aparecerem nas linhas seguintes. (**Nota:** Você deve endentar as linhas de comando com um tab, e não com espaços.)

Se o seu assembler possui o comando Make, entre com estas linhas no arquivo Dskpatch (sem extensão) e faça qualquer pequena mudança em DISP_SET.ASM. Depois digite:

```
A>MAKE DSKPATCH
```

e você verá algo assim:

```
Microsoft (R) Macro Assembler Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985. All rights reserved.

 48984 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
        link disp_set es_video cursor;

Microsoft (R) 8086 Object Linker   Version 3.05
Copyright (C) Microsoft Corp 1983, 1984, 1985. All rights reserved

Warning: no stack segment
                exe2bin disp_set disp_set.com
A>
```

Make fez um mínimo de esforço necessário para reconstruir nosso programa.

Se você não possuir uma versão recente do Macro Assembler da Microsoft, que inclui Make, poderá encontrar um outro programa muito útil chamado Symdeb (*Symbolic Debugger — Depurador Simbólico*), um ótimo substituto para o Debug. Voltaremos a ele mais adiante, mas agora vamos voltar ao Dskpatch.

Alterando Disp_set

Disp_set, conforme o deixamos, possuía uma versão de DISP_MEIO_SETOR, que usamos como rotina de teste, e a rotina principal. Agora, vamos alterar DISP_MEIO_SETOR para se tornar uma rotina normal, e poderemos chamá-la a partir de uma rotina que chamaremos de Es_disco. Nossa rotina de teste estará em Es_disco, junto com uma versão de teste de uma rotina para ler um setor do disco.

Primeiro, vamos alterar Disp_set para torná-lo um arquivo de rotinas, assim como fizemos com Es_video. Altere o END DISP_MEIO_SETOR para somente END, pois nossa rotina principal estará em Es_disco. Depois remova o comando ORG 100h de CODE_SEG, novamente porque passamos isso para um arquivo diferente.

Como planejamos ler um setor do disco para a parte da memória começando em SETOR, não há necessidade de fornecermos dados de teste. Podemos trocar todos os 16 comandos DB após SETOR por uma só linha:

```
SETOR     DB      8192 DUP (0)
```

que reserva 8192 bytes para armazenar um setor.

Mas você deve lembrar-se de que um setor ocupa somente 512 bytes. Então, para que uma área de armazenamento tão grande? Acontece que alguns discos rígidos (300 megabytes, por exemplo) usam tamanhos de setor muito grandes. Estes setores maiores não são comuns, mas queremos assegurar-nos de não aparecer um setor maior do que o espaço que foi reservado para ele em SETOR. Assim, por questão de segurança, reservamos 8192 bytes para SETOR. No restante deste livro, com exceção de SETOR, que veremos em breve, vamos assumir que os setores possuem somente 512 bytes de extensão.

O que precisamos agora é de uma nova versão de DISP_MEIO_SETOR. A antiga versão não é nada mais do que uma rotina de teste que usamos para testar DISP_LIN. Na nova versão, vamos querer dar um deslocamento ao setor, de modo que poderemos apresentar 256 bytes, começando

em qualquer lugar do setor. Entre outras coisas, isso significa que poderemos copiar a primeira metade, a última metade, ou os 256 bytes do meio (isto é, deixando de fora 128 bytes no início e 128 no final). Mais uma vez, daremos este deslocamento em DX. Eis aqui a versão nova — e última — de DISP_MEIO_SETOR em Disp_set:

Listagem 15-2. A Versão Final de DISP_MEIO_SETOR em DISP_SET.ASM

```
        PUBLIC      DISP_MEIO_SETOR
        EXTRN       ENVIA_CRLF:NEAR

;-----------------------------------------------------------------------
; Esta rotina apresenta meio setor (256 bytes)
;
;    DS:DX  Deslocamento no setor, em bytes -- deve ser multiplo de 16
;
;  Usa       DISP_LIN, ENVIA_CRLF
;-----------------------------------------------------------------------
DISP_MEIO_SETOR     PROC    NEAR
        XOR     DX,DX
        PUSH    CX
        PUSH    DX
        MOV     CX,16       ;Mostra 16 linhas
MEIO_SETOR
        CALL    DISP_LIN
        CALL    ENVIA_CRLF
        ADD     DX,16
        LOOP    MEIO_SETOR
        POP     DX
        POP     CX
        RET
        INT     20h
DISP_MEIO_SETOR     ENDP
```

Vamos prosseguir, agora, para a nossa rotina de ler um setor.

Lendo um Setor

Na primeira versão de LE_SETOR, deliberadamente ignoramos os erros, como a ausência de disco no drive. Isto não é uma boa prática, mas também não é a versão final de LE_SETOR. Não poderemos ver a manipulação de erro neste livro, mas você encontrará rotinas de manipulação de erro no disco que está disponível com o original deste livro. Por enquanto, contudo, só queremos ler um setor do disco. Eis aqui a versão de teste do arquivo ES_DISCO.ASM:

Listagem 15-3. O Novo Arquivo ES_DISCO.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        ORG     100h

        EXTRN   DISP_MEIO_SETOR:NEAR

;-----------------------------------------------------------------------
; Esta rotina le o primeiro setor do disco A e copia a primeira
; metade deste setor.
;-----------------------------------------------------------------------
LE_SETOR        PROC    NEAR
        MOV     AL,0                ;Drive A (numero 0)
        MOV     CX,1                ;Le somente um setor
        MOV     DX,0                ;Le setor numero zero
```

```
        LEA     BX,SETOR            ;Onde colocar esse setor
        INT     25h                 ;Le o setor
        POPF                        ;Descarta os flags colocados na pilha pelo DOS
        XOR     DX,DX               ;Deslocamento 0 dentro de SETOR
        CALL    DISP_MEIO_SETOR     ;Copia a primeira metade
        INT     20h                 ;Retorna ao DOS
LE_SETOR        ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC

        EXTRN   SETOR:BYTE
DATA_SEG        ENDS

        END     LE_SETOR
```

Há três instruções novas nessa rotina. A primeira:

```
LEA     BX,SETOR
```

move o *endereço*, ou deslocamento, de SETOR (desde o início de CGROUP) para o registrador BX; LEA significa *Load Effective Adress — Carregar Endereço Efetivo*. Após a instrução LEA, DS:BX conterá o endereço completo de SETOR, e o DOS utiliza esse endereço para a segunda nova instrução, a chamada INT 25h, como veremos a seguir, após mais algumas palavras sobre SETOR. (Na verdade, LEA carrega o deslocamento no registrador BX sem mexer no registrador DS; temos que nos assegurar de que DS está apontando para o segmento correto.)

SETOR não está no mesmo arquivo fonte de LE_SETOR. Está em DISP_SET.ASM. Como podemos dizer ao assembler onde ele está? Usando a pseudo-op EXTRN:

```
DATA_SEG        SEGMENT PUBLIC
        EXTRN   SETOR:BYTE
DATA_SEG        ENDS
```

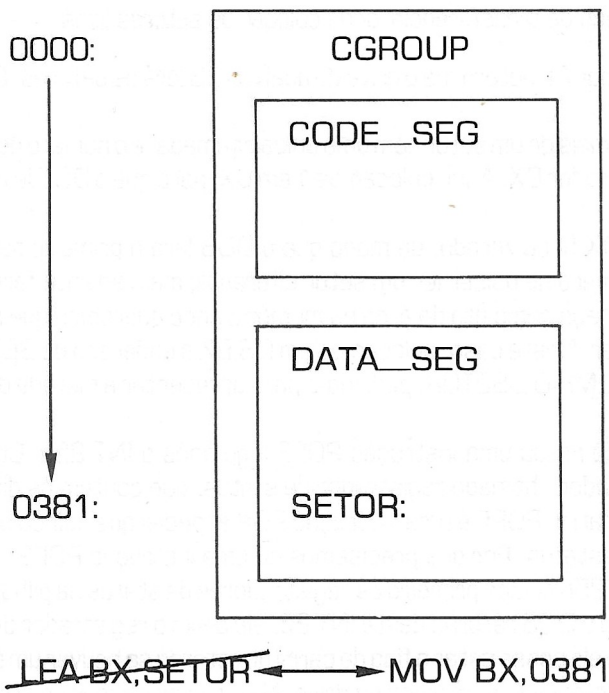


Figura 15-1. ***LEA Carrega o Endereço Efetivo.***

Este grupo de instruções diz ao assembler que SETOR está definido em DATA_SEG, que está em outro arquivo fonte, e que SETOR é uma variável de bytes (ao invés de palavras). Usaremos EXTRNs com freqüência nos capítulos seguintes; esse é o modo de usarmos as mesmas variáveis em diversos arquivos fonte. Só precisamos ter o cuidado de definir nossas variáveis em apenas um lugar.

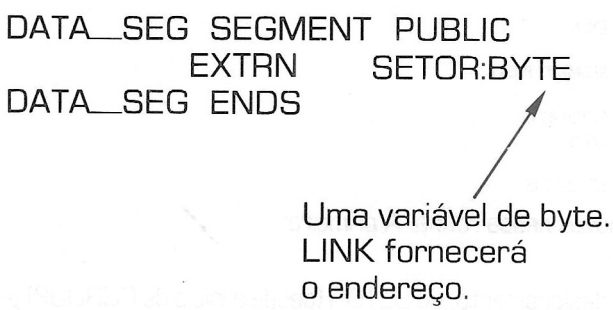


Figura 15-2. ***A Pseudo-Op EXTRN.***

Vamos retornar à instrução INT 25h. INT 25h é uma chamada de função do DOS, especialmente para ler setores do disco. Quando o DOS recebe uma chamada de INT 25h, ele usa a seguinte informação dos registradores:

AL	Número do drive (0=A, 1=B etc.)
CX	Número de setores a ser lidos de cada vez
DX	Número do primeiro setor a ser lido (o primeiro setor do disco é o 0)
DS:BX	Endereço de transferência: onde colocar os setores lidos

O número no registrador AL determina o drive do qual o DOS lerá os setores. Se AL = 0, o DOS lê no drive A.

O DOS pode ler mais de um setor com uma única chamada, e o número de setores a ser lidos encontra-se no registrador CX. Aqui, colocamos 1 em CX, para que o DOS leia somente um setor de 512 bytes.

O registrador DX ficou zerado, de modo que o DOS lerá o primeiro setor do disco. Você pode alterar esse número se quiser ler um setor diferente; mais adiante faremos isso.

DS:BX é o endereço completo da área na memória onde queremos que o DOS coloque o(s) setor(es) que ele vai ler. Neste caso, colocamos em DS:BX o endereço de SETOR, para que possamos chamar DISP_MEIO_SETOR, pedindo-o para apresentar a metade do primeiro setor do disco no drive A.

Finalmente, você notou uma instrução POPF logo após o INT 25h. Como já dissemos, o 8088 tem um registrador chamado registrador de status, que contém os diversos flags, como os flags de zero e de carry. POPF é uma instrução POP especial que coloca uma palavra da pilha para o registrador de status. Por que precisamos de uma instrução POP?

A instrução INT 25h coloca primeiro os registradores de status na pilha, e depois o endereço de retorno. Quando o DOS retorna deste INT 25, ele deixa o registrador de status na pilha. O DOS faz isso para que ele possa setar o flag de carry no retorno se houver um erro de disco, como a tentativa de se ler do drive A: sem disco no drive. Não iremos checar os erros neste livro, mas temos que retirar uma palavra da pilha — por isso precisamos da instrução POPF. (**Nota:** INT

25h, juntamente com INT 24h, que *grava* um setor no disco, são as únicas rotinas do DOS que deixam o registrador de status na pilha.)

Agora você já pode montar ES_DISCO.ASM, e montar novamente DISP_SET.ASM. Depois linkedite os quatro arquivos Es_disco, Disp_set, Es_video e Cursor, com Es_disco listado primeiro. Ou, se você tiver Make, acrescente estas duas linhas ao arquivo Dskpatch:

```
es_disco.obj        es_disco.asm
        masm es_disco.
```

e altere as últimas três linhas para:

```
es_disco.com:   es_disco.obj disp_set.obj es_video.obj cursor_obj
        link es_disco disp_set es_video cursor;
        exe2bin es_disco es_disco.com
```

Após ter criado sua versão .COM de Es_disco, você deverá ver uma tela semelhante à da Figura 15-3.

Voltaremos depois para acrescentar algo a Es_disco, mas por enquanto isso é o bastante. No capítulo seguinte, faremos uma melhor apresentação do setor, acrescentando alguns caracteres gráficos ao vídeo, além de algumas outras informações.

```
A>es_disco
EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00  δ!ÉIBM  3.1.....
02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00  .p.╨.².........
00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 0A  ...─\.3ø┐└.Ä╪3╔.
D2 79 0E 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1  ╥y.ë...î. .ê."."▒
02 8E C5 8E D5 BC 00 7C 51 FC 1E 36 C5 36 78 00  .Ä╪Ä╒╝.|Q■.6┼6x.
BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00  ┐#|╣..≤ñ.ê.,.á..
A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1  ó'.┐x.┐ø#|½æ½í..╤
E0 40 E8 80 00 E8 86 00 BB 00 05 53 B0 01 E8 AB  α@ΦÇ.Φå.╗..S░.Φ½
00 5F BE 73 01 B9 0B 00 90 F3 A6 75 62 83 C7 15  ._╛s.╣..E≤ªubâ╟.
B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B  ▒.ÉÉ≤ªuW&ïG.Öï..
00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96  ..┴H≈±Ç>q.`u.░.û
A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 1E  í..▒.╙ΦΦ;. 6..─.
6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52  o.Φ9.Φd.+≡v.Φ&.R
F7 26 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0  ≈&...╪ZδΘ=.╣..╙α
80 E4 03 74 04 FE C4 8A CC 5B 58 FF 2E 6F 01 BE  ÇΣ.t.■─è╠[X .o.╛
89 01 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18  ë.δUÉ..... .├í.

A>
```

Figura 15-3. ***Cópia da Tela de ES_DISCO.COM.***

Resumo

Agora que já temos quatro arquivos fonte, nosso programa Dskpatch está se tornando mais complicado. Neste capítulo, vimos o programa Make, que torna a tarefa mais simples, montando somente os arquivos que foram alterados.

Também escrevemos uma nova rotina, ES_DISCO. Ela está em um arquivo fonte diferente de SETOR, de modo que usamos uma definição EXTRN em ES_DISCO.ASM para informar ao assembler sobre SETOR, e fazendo-o saber que SETOR é uma variável de byte.

Também aprendemos sobre a instrução LEA, que usamos para colocar o endereço de SETOR em BX.

ES_DISCO usa um novo número de INT, INT 25h, para ler setores do disco para a memória. Usamos INT 25h para ler um setor na variável de memória, SETOR, para que pudéssemos copiá-lo na tela com DISP_MEIO_SETOR.

Também aprendemos sobre a instrução POPF, para retirar uma palavra da pilha, colocando-a no registrador de status do 8088. Usamos essa instrução para remover os flags que o DOS não retira da pilha quando retorna de INT 25h.

Nossa tela de meio setor ainda não está muito atraente, mas no próximo capítulo usaremos alguns dos caracteres gráficos presentes no IBM PC para torná-la mais agradável esteticamente.

16

MELHORANDO A APRESENTAÇÃO DO SETOR

Adicionando Caracteres Gráficos

Adicionando Endereços

Adicionando Linhas Horizontais

Adicionando Números

Resumo

Chegamos ao último Capítulo da Parte II. Tudo o que foi feito até aqui se aplica ao MS-DOS e ao 8088 (ou 8086 e outros parentes do 8088). Na Parte III começaremos a escrever rotinas específicas para o Computador Pessoal IBM e seus compatíveis mais próximos.

Antes de continuarmos, porém, usaremos este capítulo para acrescentar várias características e rotinas a Es_video. Também modificaremos DISP_LIN em Disp_set. Todos os acréscimos e modificações serão na apresentação da tela. A maioria delas será para melhorar a aparência da tela, mas uma delas adicionará novas informações: colocará números à esquerda para atuar como endereços na cópia do Debug. Vamos começar com os gráficos.

Adicionando Caracteres Gráficos

O Computador Pessoal IBM possui uma série de caracteres de desenho de linha que servem para desenhar quadros em torno de uma informação na tela. Desenharemos um quadro em torno da nossa cópia hexa, e outro em torno da cópia ASCII. Esta mudança não requer muito trabalho mental, mas físico.

Entre com as seguintes definições próximas ao topo do arquivo DISP_SET.ASM, entre o ASSUME e o primeiro SEGMENT, deixando uma ou duas linhas em branco antes e depois dessas definições:

Listagem 16-1. Acréscimo no Topo de DOSP-SET.ASM.

```
;--------------------------------------------------------------------------;
; Caracteres graficos para o quadro do setor.                              ;
;--------------------------------------------------------------------------;
BARRA_VERT      EQU     0BAh
BARRA_HOR       EQU     0CDh
SUP_ESQ         EQU     0C9h
SUP_DIR         EQU     0BBh
INF_ESQ         EQU     0C8h
INF_DIR         EQU     0BCh
T_ALTO          EQU     0CBh
T_BAIXO         EQU     0CAh
TRACO_ALTO      EQU     0D1h
TRACO_BAIXO     EQU     0CFh
```

Estas são as definições dos caracteres gráficos. Note que colocamos um zero antes do número hexa para que o assembler saiba que eles são números, e não rótulos.

Poderíamos facilmente ter escrito números em hexa diretamente no trecho do código, mas as definições tornam a rotina mais fácil de entender e modificar. Por exemplo, compare as duas instruções a seguir:

```
MOV     DL,BARRA_VERT
MOV     DL,0BAh
```

A maioria das pessoas acha que a primeira instrução é bem mais clara.

Agora, veja a rotina DISP_LIN para separar as diferentes partes do vídeo com uma barra vertical, caractere 186 (0BAh). Como antes, os acréscimos são mostrados em negrito:

Listagem 16-2. Mudanças em DISP_LIN, de DISP_SET.ASM

```
DISP_LIN        PROC    NEAR
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                   ;Deslocamento mais util em BX
```

```
                                ;Separa a escrita
        MOV   DL,' '
        CALL  IMP_CAR
        MOV   DL,BARRA_VERT     ;Desenha lado esquerdo do quadro
        CALL  IMP_CAR
        MOV   DL,' '
        CALL  IMP_CAR
                                ;Agora imprime 16 bytes
        MOV   CX,16             Copia 16 bytes
        PUSH  BX                Salva o deslocamento para LOOP_ASCII
LOOP_HEXA:
        MOV   DL,SETOR[BX]      Pega 1 byte
        CALL  IMP_HEXA          Copia o byte em hexa
        MOV   DL,' '            Espaco entre os numeros
        CALL  IMP_CAR
        INC   BX
        LOOP  LOOP_HEXA
        MOV   DL,BARRA_VERT     ;Separa a escrita
        CALL  IMP_CAR
          MOV   DL,' '
          CALL  IMP_CAR

          MOV   CX,16
          POP   BX                      ;Volta com o deslocamento no setor
LOOP_ASCII:
          MOV   DL,SETOR[BX]
          CALL  IMP_CAR
          INC   BX
          LOOP  LOOP_ASCII

        MOV   DL,' '            ;Desenha lado direito do quadro
        CALL  IMP_CAR
        MOV   DL,BARRA_VERT
        CALL  IMP_CAR

        POP                                                          DX
        POP   CX
        POP   BX
        RET
DISP_LIN        ENDP
```

Monte esta nova versão de Disp_set e linkedite os quatro arquivos (lembre-se de colocar Es_disco logo após o comando LINK, em primeiro lugar). Você verá umas barras verticais separando a tela em duas partes, como podemos ver na Figura 16-1.

```
A> es_disco
EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00 | δ!ÉIBM  3.1.....
02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 | .p.╨.².........
00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 0A | ...─\.3ø╕└.Ä╪3╔.
D2 79 0E 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1 | ╥y.ë...î. .ê."."▒
02 8E C5 8E D5 BC 00 7C 51 FC 1E 36 C5 36 78 00 | .Ä╪Ä╒╝.|Q■.6┼6x.
BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00 | ┐#|╣..≤ñ.ê.,.á..
A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1 | ó'.┐x.╕#|½æ½í..╤
E0 40 E8 80 00 E8 86 00 BB 00 05 53 B0 01 E8 AB | α@╪Ç.╪å.╗..S▒.╪½
00 5F BE 73 01 B9 0B 00 90 F3 A6 75 62 83 C7 15 | ._╛s.╣..≥ªubâ╟.
B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B | ▒.ÉÉ≤ªuW&ïG.Öï..
00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96 | ..┴H≈±Ç>q.`u.▒.û
A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 1E | í..▒.╙╪╪;. 6..─.
6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52 | o.╪9.╪d.+≡v.╪&.R
F7 26 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0 | ≈&...╪Zδθ=.╣..╙α
80 E4 03 74 04 FE C4 8A CC 5B 58 FF 2E 6F 01 BE | ÇΣ.t.■─è╠[X .o.╛
89 01 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18 | ë.δUÉ..... .├í.

A>
```

Figura 16-1. ***Adicionando Barras Verticais***

Adicionando Endereços

Vamos agora tentar algo mais desafiador: vamos colocar os endereços hexa ao lado esquerdo da tela. Esses números serão o deslocamento do início do setor, de modo que o primeiro número será 00, o seguinte 10, e depois 20, e assim por diante.

O processo é muito simples, pois já temos a rotina IMP_HEXA para imprimir os números em hexa. Mas temos um problema ao lidar com setores de 512 bytes: IMP_HEXA imprime somente números hexa de dois dígitos, e precisamos de três dígitos para números maiores do que 255.

Eis a solução. Como os números estarão entre 0 e 511 (0h a 1FFh), o primeiro dígito será ou um espaço (como em BCh) se o número for menor do que 100h, ou será um 1. Assim, se o número for maior do que 255 simplesmente imprimiremos um, seguido pelo menor byte do número hexa. Caso contrário, imprimiremos primeiro um espaço. Estes são os acréscimos em DISP_LIN que imprimirão os três dígitos do número hexa:

Listagem 16-3. Acréscimos em DISP_LIN, de DISP_SET.ASM

```
DISP_LIN        PROC    NEAR
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX           ;Deslocamento mais util em BX
        MOV     DL,''
                                ;Imprime deslocamento em hexa
        CMP     BX,100h         ;Primeiro digito = 1?
        JB      IMP_UM          ;Nao; espaco em branco ja em DL
        MOV     DL,'1'          ;Sim, coloca "1" em DL para saida hexa
IMP_UM:
        CALL    IMP_CAR
        MOV     DL,BL           ;Copia byte baixo em DL para saida hexa
        CALL    IMP_HEXA
                                ;Separa a escrita
        MOV     DL,'
        CALL    IMP_CAR
        MOV     DL,BARRA_VERT   ;Desenha lado esquerdo do quadro
```

Você pode ver o resultado na Figura 16-2.

```
A>es_disco
 00 ‖ EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00 ‖ δ!ÉIBM  3.1.....
 10 ‖ 02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 ‖ .p.╨.²..........
 20 ‖ 00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 0A ‖ ...-\.3ø┐└.Ä╪3╔.
 30 ‖ D2 79 0E 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1 ‖ ╥y.ë...î. .ê.".▒
 40 ‖ 02 8E C5 8E D5 BC 00 7C 51 FC 1E 36 C5 36 78 00 ‖ .Ä╞Ä╒╝.¦Qⁿ.6╞6x.
 50 ‖ BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00 ‖ ┐#¦╣..≤ñ.ê.,.á..
 60 ‖ A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1 ‖ ó'.┐x.┐#¦½æ½í..╤
 70 ‖ E0 40 E8 80 00 E8 86 00 BB 00 05 53 B0 01 E8 AB ‖ «@ΦÇ.Φå.╗..S▒.Φ½
 80 ‖ 00 5F BE 73 01 B9 0B 00 90 F3 A6 75 62 83 C7 15 ‖ ._╝s.╣..É≤ªubâ╟.
 90 ‖ B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B ‖ ▒.ÉÉ≤ªuW&ïG.Öï..
 A0 ‖ 00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96 ‖ ..┴H≈±Ç>q.`u.▒.û
 B0 ‖ A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 1E ‖ í..▒.╙ΦΦ;. 6..-.
 C0 ‖ 6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52 ‖ o.Φ9.Φd.+≡v.Φ&.R
 D0 ‖ F7 26 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0 ‖ ≈&...╪ZδΘ=.╣..╙«
 E0 ‖ 80 E4 03 74 04 FE C4 8A CC 5B 58 FF 2E 6F 01 BE ‖ ÇΣ.t.▪-è╟[X .o.╝
 F0 ‖ 89 01 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18 ‖ ë.δUÉ...... .├í.

A>
```

Figura 16-2. ***Adicionando Números à Esquerda***

Estamos chegando perto de uma tela completa. Porém, nossa listagem não está centralizada na tela. Precisamos movê-la para a direita em cerca de três espaços. Vamos fazer essa pequena mudança, e depois teremos nossa versão final de DISP_LIN.

Poderíamos fazer isso chamando IMP_CAR três vezes com um espaço em branco, mas não dessa vez. Ao invés disso, vamos criar outra rotina, chamada IMP_CAR_N_VEZES, junto com Es_video. Como o nome indica, esta rotina imprime um caractere N vezes. Isto é, colocamos o número N no registrador CX e o código do caractere em DL, e depois chamamos IMP_CAR_N_VEZES para imprimir N cópias do caractere cujo código ASCII nós colocamos em DL. Assim, poderemos imprimir três espaços colocando 3 em CX e 20h (o código ASCII do espaço) em DL.

Eis a rotina que deve ser acrescentada a ES_VIDEO.ASM:

Listagem 16-4. Acrescente esta Rotina em ES_VIDEO.ASM

```
        PUBLIC  IMP_CAR_N VEZES

;-----------------------------------------------------------------------;
; Esta rotina imprime mais de uma copia de um caractere                 ;
;                                                                       ;
;    DL     Codigo do caractere                                        ;
;    CX     Numero de vezes a imprimir                                 ;
;                                                                       ;
; Usa:       IMP_CAR                                                    ;
;-----------------------------------------------------------------------;
IMP_CAR_N_VEZES     PROC    NEAR
    PUSH    CX
N_VEZES:
    CALL    IMP_CAR
    LOOP    N_VEZES
    POP     CX
    RET
IMP_CAR_N_VEZES     ENDP
```

Você pode ver a simplicidade dessa rotina, pois já temos IMP_CAR. Se você está querendo saber por que nos preocupamos em escrever uma rotina separada para algo tão simples, a resposta é que o programa final ficará muito mais claro se chamarmos uma rotina IMP_CAR_N_VEZES do que se copiarmos essa rotina dentro do próprio programa. Além disso, faremos uso dessa mesma rotina em outros lugares também.

Eis as mudanças em DISP_LIN para acrescentar três espaços à esquerda de cada linha da tela. Faça as alterações em DISP_SET.ASM:

```
        PUBLIC  DISP_LIN
        EXTRN   IMP_HEXA:NEAR
        EXTRN   IMP_CAR:NEAR
        EXTRN   IMP_CAR_N_VEZES:NEAR

;-----------------------------------------------------------------------;
; Esta rotina mostra uma linha de dados, ou 16 bytes,                   ;
; primeiro em hexa, e depois em ASCII                                   ;
;                                                                       ;
;    DS:DX   Deslocamento no setor, em bytes.                           ;
;                                                                       ;
; Usa:       IMP_CAR, IMP_HEXA, IMP_CAR_N_VEZES                         ;
; Le:        SETOR                                                      ;
;-----------------------------------------------------------------------;
DISP_LIN        PROC    NEAR
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX               ;Deslocamento mais util em BX
        MOV     DL,' '
        MOV     CX,3                ;Imprime 3 espaços antes da linha
        CALL    IMP_CAR_N_VEZES
```

```
                                        ;Imprime deslocamento em hexa
        CMP     BX,100h                 ;Primeiro digito = 1?
        JB      IMP__UM                 ;Nao, espaco em branco ja em DL.
        MOV     DL,'1'                  ;Sim, coloca "1" em DL para saida hexa.
IMP__UM:
```

Fizemos mudanças em três lugares. Primeiro, tivemos que incluir um comando EXTRN para IMP__CAR__N__VEZES, pois a rotina está em Es__video, e não nesse arquivo. Também alteramos o bloco de comentários, para mostrar que usamos essa nova rotina. Nossa terceira mudança, as duas linhas que usam IMP__CAR__N__VEZES, é muito simples e não precisa de explicação.

Execute essa nova versão do programa, e veja como a listagem do setor está agora centralizada na tela. A seguir, vamos acrescentar novas características a nossa tela — as linhas de cima e de baixo dos quadros.

Adicionando Linhas Horizontais

O acréscimo de linhas horizontais a nossa tela não é tão simples quanto parece, pois temos alguns casos especiais em que temos que pensar. Temos os extremos, onde as linhas devem fazer o contorno para a vertical, e temos também as formas de T que estão na divisão entre as duas janelas da tela, hexa e ASCII.

Poderíamos escrever uma longa lista de instruções (com IMP__CAR__N__VEZES) para criar as linhas horizontais, mas não vamos fazer dessa forma. Temos um modo mais curto. Apresentaremos uma outra rotina, chamada IMP__FORMA, que imprimirá uma forma na tela. Depois, tudo do que precisaremos é de uma área na memória para conter a descrição de cada forma. Usando esta nova rotina, podemos também facilmente acrescentar pequenos traços para subdividir a janela hexadecimal, como você verá ao final desta seção.

IMP__FORMA usa duas instruções inteiramente novas, LODSB e CLD. Falaremos sobre elas depois de vermos mais um pouco sobre a rotina IMP__FORMA e como descrevemos uma forma. Agora, entre com essa rotina em ES__VIDEO.ASM:

Listagem 16-5. Acrescente esta Rotina em ES__VIDEO.ASM

```
        PUBLIC  IMP__FORMA

;-----------------------------------------------------------------------
; Esta rotina imprime uma linha na tela, baseada nos dados
; de formato
;
;       DB      caractere, numero de vezes a imprimir  ,0
;         x     significa que x pode ser repetido qualquer numero de vezes
;       DB      Endereço do comando de dados acima.
;
; Usa:          IMP__CAR__N__VEZES
;-----------------------------------------------------------------------
IMP__FORMA      PROC    NEAR
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSHF                           ;Salva o flag de direcao
        CLD                             ;Seta o flag direcao p/incremento
        MOV     SI,DX                   ;Deslocamento --> SI para LODSB
LOOP__FORMA:
        LODSB                           ;Pega o dado e coloca em AL
        OR      AL,AL                   ;E o fim da lista (0h)?
        JZ      AL,AL                   ;Sim, retorna
        MOV     DL,AL                   ;Nao, prepara para imprimir N vezes
        LODSB                           ;Apanha o contador para AL
        MOV     CL,AL                   ;Coloca em CX para IMP__CAR__N__VEZES
        XOR     CH,CH                   ;Zera byte alto de CX
```

```
        CALL    IMP_CAR_N_VEZES
        JMP     LOOP_FORMA
FIM_FORMA:
        POPF                            ;Restaura flag de direcao
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
IMP_FORMA       ENDP
```

Antes de vermos como essa rotina funciona, vejamos como imprimir os dados para as formas. Colocaremos os dados para o formato da linha de cima em Disp_set, que é onde iremos usá-lo. Para isso, colocaremos uma outra rotina, chamada INIC_DISP_SET, para inicializar a apresentação do setor imprimindo a metade do setor, e depois modificaremos LE_SETOR para que chame nossa rotina INIC_DISP_SET.

Primeiro, coloque os seguintes dados após SETOR (em DISP_SET.ASM), dentro do segmento de dados:

Listagem 16-6 Acréscimos a DISP_SET.ASM

```
FORMA_LIN_SUP       LABEL     BYTE
        DB      DB ';?
        DB      SUP_ESQ,1
        DB      BARRA_HOR,12
        DB      TRACO_ALTO,1
        DB      BARRA_HOR,11
        DB      TRACO_ALTO,1
        DB      BARRA_HOR,11
        DB      TRACO_ALTO,1
        DB      BARRA_HOR,12
        DB      T_ALTO,1
        DB      BARRA_HOR,18
        DB      SUP_DIR,1
        DB      0
FORMA_LIN_INF
        DB      DB '?
        DB      SUP_ESQ,1
        DB      BARRA_HOR,12
        DB      TRACO._BAIXO,1
        DB      BARRA_HOR,11
        DB      TRACO_BAIXO,1
        DB      BARRA_HOR,11
        DB      TRACO_BAIXO,1
        DB      BARRA_HOR,12
        DB      T_BAIXO,1
        DB      BARRA_HOR,18
        DB      INF_DIR,1
```

Cada comando DB contém parte dos dados de uma linha. O primeiro byte é o caractere a ser impresso; o segundo byte diz a IMP_FORMA quantas vezes ele deve repetir esse caractere. Por exemplo, começamos a linha de cima com sete espaços em branco, seguidos por um caractere de canto superior esquerdo, seguido por doze caracteres de barra horizontal, e assim por diante, o último DB é simplesmente 0, o que marca o final da forma.

Vamos continuar nossas alterações e ver o resultado, antes de comentarmos o funcionamento interno de IMP_FORMA. Aqui está a versão de teste de INIC_DISP _SET. Esta rotina imprime a forma da linha de cima, a apresentação das linhas do setor, e finalmente a forma da linha de baixo. Coloque-o no arquivo DISP_SET.ASM, logo antes de DISP_MEIO _SETOR:

Listagem 16-7. Acrescente esta Rotina em DISP_SET.ASM

```
        PUBLIC  INIC_DISP_SET
        EXTRN   IMP_FORMA:NEAR, ENVIA_CRLF:NEAR

;-------------------------------------------------------------------------
; Esta rotina inicializa a apresentacao do setor.
;
; Usa:      IMP_FORMA, ENVIA_CRLF, DISP_MEIO_SETOR
; Le:       FORMA_LIN_SUP, FORMA_LIN_INF
;-------------------------------------------------------------------------
INIC_DISP_SET   PROC    NEAR
        PUSH    DX
        LEA     FORMA_LIN_SUP
        CALL    IMP_FORMA
        CALL    ENVIA_CRLF
        XOR     DX,DX                   ;Inicia no comeco do setor
        CALL    DISP_MEIO_SETOR
        LEA     FORMA_LIN_INF
        CALL    IMP_FORMA
        POP     DX
        RET
INIC_DISP_SET   ENDP
```

Usamos a instrução LEA para carregar um endereço no registrador DX, para que IMP_FORMA saiba onde encontrar os dados das duas linhas horizontais.

Finalmente, precisamos fazer uma pequena mudança em SE_SETOR no arquivo ES_DISCO.ASM, para chamar INIC_DISP_SETOR ao invés de DISP_MEIO_SETOR, para que o quadro completo possa ser desenhado em torno dos bytes de dados do setor:

Listagem 16-8. Alterações em LE_SETOR, de ES_DISCO.ASM

```
        EXTRN   INIC_DISP_SET:NEAR

;-------------------------------------------------------------------------
; Esta rotina le o primeiro setor do disco A e copia a primeira
; metade deste setor.
;-------------------------------------------------------------------------
LE_SETOR        PROC    NEAR
        MOV     AL,0                    ;Drive A (numero 0)
        MOV     CX,1                    ;Le somente um setor
        MOV     DX,0                    ;Le setor numero zero
        LEA     BX,SETOR                ;Onde colocar esse setor
        INT     25h                     ;Le o setor
        POPF                            ;Descarta os flags colocados na pilha pelo DOS
        XOR     DX,DX                   ;Deslocamento 0 dentro de SETOR
        CALL    INIC_DISP_SET           ;Copia a primeira metade
        INT     20h                     ;Retorna ao DOS
LE_SETOR        ENDP
```

Isso é tudo o que precisamos imprimir nas linhas de cima e de baixo da apresentação do setor. Monte e linkedite todos esses arquivos (lembre-se de montar os três arquivos que alteramos), execute o resultado por meio de Exe2bin, e depois rode o programa (Es_disco). A Figura 16-3 mostra a saída que temos agora.

Vejamos agora como IMP_FORMA funciona. Como dissemos, ele usa duas novas instruções. LODSB significa *Load String Byte — Carrega Byte de String —*, e é uma das *instruções de string*: instruções feitas especialmente para trabalharem com strings de caracteres. Não é exatamente isso o que vamos fazer aqui, mas o 8088 não se incomodará se usarmos essa instrução emprestada para trabalharmos com números ao invés de caracteres, de modo que LODSB serve bem para nossos propósitos aqui.

LODSB move (carrega) um único byte para o registrador AL a partir da posição de memória dada por DS:SI, um par de registradores que ainda não tínhamos usado. Todos os registradores de

```
A>es  disco

00  EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00   δ!ÉIBM  3.1.....
10  02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00   .p.╨.².........
20  00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 0A   ...─\.3φ╕└.Ä╪3╔.
30  D2 79 0E 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1   ╥y.ë...î. .ê."."▒
40  02 8E C5 8E D5 BC 00 7C 51 FC 1E 36 C5 36 78 00   .Ä╪Ä╒╝.|Q■.6┼6x.
50  BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00   ┐#|╣..≤ñ.ê.,.á..
60  A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1   ó'.┐x.┐#|½æ½í..╤
70  E0 40 E8 80 00 E8 86 00 BB 00 05 53 B0 01 E8 AB   α@Φ╟.Φå.╗..S░.Φ½
80  00 5F BE 73 01 B9 0B 00 90 F3 A6 75 62 83 C7 15   ._╛s.╣..É≤ªubâ╟.
90  B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B   ▒.ÉÉ≤ªuW&ïG.Öï..
A0  00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96   ..┴H≈±Ç>q.`u.░.û
B0  A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 1E   í..▒.╙ΦΦ;. 6..─.
C0  6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52   o.Φ9.Φd.+≡v.Φ&.R
D0  F7 26 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0   ≈&...╪Zδθ=.╣..╙α
E0  80 E4 03 74 04 FE C4 8A CC 5B 58 FF 2E 6F 01 BE   ÇΣ.t.■─è╠[X .o.╛
F0  89 01 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18   ë.δUÉ..... .├í.

A>
```

Figura 16-3. ***A Tela com os Quadros Fechados.***

segmento em nosso arquivo .COM são inicializados com nosso único segmento, CGROUP, de modo que DS já possui o seu valor inicializado. E antes da instrução LODSB, movemos o deslocamento para SI com a instrução MOV SI,DX (lembre-se de que DX está sendo sempre usado para guardar o deslocamento a ser passado para outras rotinas).

A instrução LODSB é algo parecido com a instrução MOV, porém é mais poderosa. Com uma instrução LODSB, o 8088 move um byte para o registrador AL, e depois ou incrementa ou decrementa o registrador SI. O incremento do registrador SI aponta o byte seguinte na memória; e o decremento aponta o byte anterior.

O primeiro (incrementando) é exatamente o que queremos fazer. Queremos ler a forma, um byte de cada vez, começando do início dos índices; e é isso o que a instrução LODSB faz. Mas como o 8088 vai saber se queremos incrementar ou decrementar o registrador SI? Simples, com a outra nova instrução, CLD (*"Clear Direction Flag" — Zera Flag de Direção*). Se tivéssemos setado o flag de direção (ao invés de o termos zerado), o registrador SI seria decrementado, e LODSB realizaria a tarefa ao contrário. Usaremos a instrução LODSB em alguns outros lugares em Dskpatch, sempre com o flag de direção zerado, para que incremente sucessivamente o endereço dos bytes lidos.

Além das instruções LODSB e CLD, note que também usamos as instruções PUSHF e POPF para salvar e recuperar o registrador de flags. Fizemos isso para o caso de decidirmos depois usar o flag de direção em uma rotina que chame IMP_FORMA.

Adicionando Números

Estamos quase terminando a Parte II deste livro. Criaremos só mais uma rotina, e depois passaremos à Parte III, para vermos coisas maiores e melhores.

Agora veja nossa versão da tela, e note que ela ainda falta uma linha de números no alto. Tais números — 00, 01, 02 etc. — nos permitiriam olhar para a coluna abaixo e encontrar qualquer byte com referência ao seu endereço. Assim sendo, vamos escrever uma rotina para imprimir uma seqüência de números no alto. Acrescente esta rotina, IMP_INDICES, no arquivo DISP_SET.ASM, logo após a rotina INIC_DISP_SET:

Listagem 16-9. Acrescente esta Rotina em DISP_SET.ASM

```
        EXTRN   IMP_CAR_N_VEZES:NEAR, IMP_HEXA:NEAR, IMP_CAR:NEAR
        EXTRN   IMP_DIG_HEXA:NEAR, ENVIA_CRLF:NEAR

;-------------------------------------------------------------------------
; Esta rotina escreve os indices (0 a F) no topo da apresentacao
; do meio setor.
;
; Usa:      IMP_CAR_N_VEZES, IMP_HEXA, IMP_CAR,
;           IMP_DIG_HEXA, ENVIA_CRLF
;-------------------------------------------------------------------------
IMP_INDICES         PROC    NEAR
        PUSH    CX
        PUSH    DX
        MOV     DL,' '                  ;Imprime 9 espacos para o lado esquerdo
        MOV     CX,9
        CALL    IMP_CAR_N_VEZES
        XOR     DH,DH                   ;Inicia com 0
LOOP_NR_HEXA:
        MOV     DL,DH
        CALL    IMP_HEXA
        MOV     DL,' '
        CALL    IMP_CAR
        INC     DH
        CMP     DH,10h                  ;Terminou?
        JB      LOOP_NR_HEXA

        MOV     DL,' '                  ;Imprime os digitos sobre a janela ASCII
        MOV     CX,2
        CALL    IMP_CAR_N_VEZES
        XOR     DL,DL
LOOP_DIG_HEXA:
        CALL    IMP_DIG_HEXA
        INC     DL
        CMP     DL,10h
        JB      LOOP_DIG_HEXA
        CALL    ENVIA_CRLF
        POP     DX
        POP     CX
        RET
IMP_INDICES     ENDP
```

Modifique INIC_DISP_SET (também em DISP_SET.ASM) como listado a seguir, de modo que chame a IMP_INDICES antes de imprimir o corpo da listagem do meio-setor:

Listagem 16-10. Mudanças em INIC_DISP_SET, de DISP_SET.ASM

```
;-------------------------------------------------------------------------
; Usa:      IMP_FORMA, ENVIA_CRLF, DISP_MEIO_SETOR
;           IMP_INDICES
; Le:       FORMA_LIN_SUP, FORMA_LIN_INF
;-------------------------------------------------------------------------
INIC_DISP_SET   PROC    NEAR
        PUSH    DX
        CALL    IMP_INDICES
        LEA     DX,FORMA_LIN_SUP
        CALL    IMP_FORMA
        CALL    ENVIA_CRLF
        XOR     DX,DX                   ;Inicia no comeco do setor
        CALL    DISP_MEIO_SETOR
        LEA     DX,FORMA_LIN_INF
        CALL    IMP_FORMA
        POP     DX
        RET
INIC_DISP_SET   ENDP
```

Agora já temos uma apresentação completa do meio setor, como você pode ver pela Figura 16-4.

Ainda há algumas diferenças entre esta tela e a versão final. Iremos alterar IMP_CAR de modo que imprima todos os 256 caracteres que o IBM PC pode apresentar, e depois limparemos a tela e centralizaremos estas listagens verticalmente, usando as rotinas do ROM BIOS, internas ao IBM PC. Faremos isso em seguida.

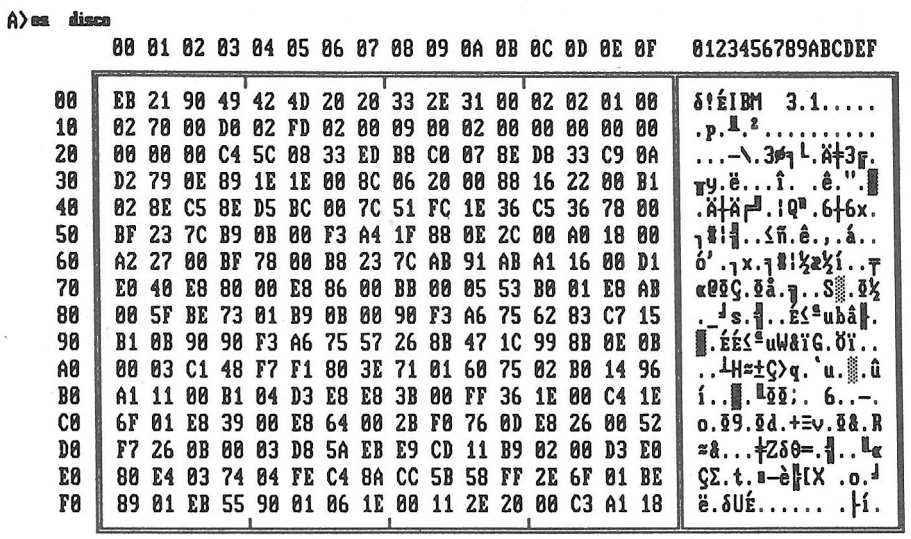


Figura 16-4. ***A Apresentação Completa de Meio Setor.***

Resumo

Já fizemos muita coisa em direção ao nosso programa Dskpatch, criando novas rotinas, alterando antigas, movendo de um arquivo fonte para outro etc. Daqui para frente, se você se perder em relação ao que estamos fazendo, veja a listagem completa de Dskpatch no Apêndice B. Aquela listagem é a versão final, mas provavelmente você verá trechos que o ajudarão a encontrar-se.

A maioria das nossas alterações aqui não apresentou novos truques, mas apenas trabalho estético. Aprendemos, porém, duas novas instruções: LODSB e CLD. LODSB é uma das instruções de string que nos permite fazer muito trabalho com somente uma instrução. Usamos LODSB em IMP_FORMA para ler bytes consecutivos de uma tabela de formas, sempre movendo o novo byte para o registrador AL. CLD zera o flag de direção, que define a direção para incremento. Cada instrução LODSB seguinte moverá o próximo byte da memória para o registrador AL.

Na próxima parte deste livro iremos aprender sobre as rotinas do ROM BIOS do seu PC. Elas economizarão muito o nosso tempo.

PARTE III

O ROM BIOS DO IBM PC

17

AS ROTINAS DO ROM BIOS

VIDEO_IO, as Rotinas do ROM BIOS

Movendo o Cursor

Ampliando o Uso de Variáveis

Imprimindo o Cabeçalho

Resumo

Dentro do seu IBM PC, existem alguns "chips" (*Circuitos Integrados*), conhecidos como ROMs (*Read Only Memory — Memória Somente de Leitura*). Uma dessas ROMs contém uma série de rotinas, que fornecem os passos básicos para que o 8088 realize uma série de tarefas de entrada e saída nos mais diversos equipamentos instalados em seu computador. Por essa ROM fornecer rotinas para a execução de entrada e saída em um nível muito baixo (nível de máquina), ela é geralmente conhecida como ROM BIOS (*Basic Input Output System — Sistema Básico para Entrada e Saída*). O DOS usa o ROM BIOS para atividades como enviar caracteres para a tela, ler caracteres do teclado, ler e gravar em disco, e temos a capacidade de usar essas rotinas em nossos programas.

Iremos concentrar-nos nas rotinas do BIOS que estaremos usando em nosso programa Dskpatch, que possui uma série de funções que não poderíamos realizar sem o trabalho direto do hardware — seria uma tarefa muito difícil.

VIDEO_IO, as Rotinas do ROM BIOS

Os elementos do ROM BIOS são referenciados como rotinas, para que as diferencie das "procedures". Até aqui, usamos ambos os termos como uma coisa só. Procuraremos usar agora o termo procedure para o código chamado através de instruções CALL, e chamaremos de rotina quando for utilizada com instruções INT. Usaremos, por exemplo, uma instrução INT 10h, para chamar as rotinas de I/O (Entrada e Saída), assim como usamos INT 21h para chamar as rotinas no DOS.

Especificamente, INT 10h chama a rotina VIDEO_IO no ROM BIOS. Outros números chamam outras rotinas, mas não iremos vê-los aqui; VIDEO_IO possui todas as funções de que precisamos fora do DOS. (Só para sua informação, o DOS chama uma das outras rotinas do ROM BIOS quando pedimos um setor do disco.)

Neste capítulo, usaremos as rotinas do ROM BIOS para acrescentar novas procedures a Dskpatch: uma para limpar a tela, e outra para movermos o cursor para qualquer posição que quisermos. Ambas as funções são muito úteis, mas nenhuma está disponível diretamente pelo DOS; por isso, usaremos as rotinas do ROM BIOS para fazer esse trabalho. Depois, veremos coisas mais interessantes que podemos fazer com essas rotinas em ROM, mas vamos começar usando INT 10h para limpar a tela antes de apresentar a metade do setor.

A instrução INT 10h é nosso ponto de entrada para uma série de funções diferentes. Lembre-se de que, quando usamos a instrução INT 21h para o DOS, selecionamos uma função em particular colocando um número de função no registrador AH. Selecionamos uma função de VIDEO_IO também da mesma forma: colocando o número apropriado da função no registrador AH (há uma lista completa das funções na Tabela 17-1).

Tabela 17-1. Funções de INT 10h

(AH)=0	**Seta o modo de vídeo.** O registrador AL contém o número do modo.
	MODO TEXTO
(AL)=0	40 por 25, modo preto e branco
(AL)=1	40 por 25, colorido
(AL)=2	80 por 25, preto e branco
(AL)=3	80 por 25, colorido
(AL)=7	80 por 25, adaptador de vídeo monocromático

MODO GRÁFICO

(AL)=4 320 por 200, colorido
(AL)=5 320 por 320, preto e branco
(AL)=6 640 por 200, preto e branco

(AH)=1 **Seta o tamanho do cursor.**

(CH) Linha de varredura inicial do cursor. A linha do alto é zero tanto no vídeo monocromático quanto no colorido, enquanto a linha de baixo é 7 para o colorido e 13 para o monocromático. Faixa válida: de 0 a 31.

(CL) Última linha de varredura do cursor. Na inicialização do computador, o adaptador colorido define CH = 6 e CL = 7. Para o vídeo monocromático: CH = 11 e CL = 12.

(AH)=2 **Seta a posição do cursor.**

(DH,DL) Linha, coluna da nova posição do cursor; o canto superior esquerdo é (0,0).

(BH) Número da página. Esse é o número da página de vídeo. O adaptador de gráficos coloridos possui espaço para diversas páginas de vídeo, mas a maioria dos programas usa a página 0.

(AH)=3 **Lê a posição do cursor.**

(BH) Número da página.
Saída (DH,DL) Linha, coluna do cursor
(CH,CL) Tamanho do cursor

(AH)=4 **Lê a posição da caneta ótica.**
(Veja o Manual de Referência Técnica.)

(AH)=5 **Seleciona página de vídeo ativa.**

(AL) Novo número de página (de 0 a 7 para os modos de 0 a 1; de 0 a 3 para modos 2 e 3).

(AH)=6 **Rolamento de tela.**

(AL) Número de linhas a colocar no final da janela. Normalmente se coloca uma linha. Coloque zero para limpar a janela inteira.

(CH,CL) Linha, coluna do canto superior esquerdo da janela.

(DH,DL) Linha, coluna do canto inferior direito da janela.

(BH) Atributo de vídeo a usar para linhas em branco.

(AH)=7 **Abaixamento da tela.**

Igual à função de rolamento (função 6), mas as linhas são deixadas em branco no alto.

(AH)=8 **Lê atributo e caractere na posição do cursor.**

(BH)	Página de vídeo (somente modo texto)
(AL)	Caractere lido
(AH)	Atributo do caractere lido (somente modo texto)

(AH)=9 **Imprime atributo e caractere na posição do cursor.**

(BX)	Página de vídeo (somente modo texto)
(CX)	Número de vezes a imprimir o caractere e o atributo na tela
(AL)	Caractere a imprimir
(BL)	Atributo a imprimir

(AH)=10 **Imprime caractere na posição do cursor** (com atributo normal).

(BH)	Página de vídeo
(CX)	Número de vezes a imprimir o caractere
(AL)	Caractere a imprimir

(AH)=11 a 13 **Diversas funções gráficas.** (Veja os detalhes no Manual de Referência Técnica)

(AH) = 14 **Imprime teletipo.** Imprime um caractere na tela e move o cursor para a próxima posição.

(AL)	Caractere a imprimir
(BL)	Cor do caractere (somente modo gráfico)
(BH)	Página de vídeo (modo texto)

(AH)=15 **Retorna estado atual do vídeo.**

(AL)	Modo de vídeo atualmente setado
(AH)	Número de caracteres por linha
(BH)	Páginas de vídeo ativas

Usaremos a função número 6 de INT 10h, ROLAMENTO DA PÁGINA ATIVA, para limpar a tela. Não vemos a tela, mas essa função também serve para limpar a tela. Eis a rotina. Entre-a no arquivo CURSOR.ASM:

Listagem 17-1. Acrescente esta Procedure em CURSOR.ASM

```
        PUBLIC  LIMPA_TELA

;-------------------------------------------------------------------------;
; Esta rotina limpa a tela inteira.                                       ;
;-------------------------------------------------------------------------;
LIMPA_TELA      PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        XOR     AL,AL       ;Limpa a janela inteira.
        XOR     CX,CX       ;Canto sup. esq. (0,0)
        MOV     DH,24       ;Ultima linha = 24
        MOV     DL,79       ;Coluna direita = 79
        MOV     BH,7        ;Atributo normal
        MOV     AH,6        ;Chama funcao 6
        INT     10h         ;Limpa a janela
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LIMPA_TELA      ENDP
```

Aparentemente, a função 6 de INT 10h precisa de muita informação para somente limpar a tela. Esta função é muito mais poderosa: ela pode limpar qualquer parte retangular da tela — *janela*, como é chamada. Para apagarmos uma tela inteira, temos que definir a primeira e a última linhas como 0 e 24, e definir as colunas como 0 e 79. A rotina que estamos usando aqui pode também limpar a tela inteira em branco (para uso em caracteres pretos), ou em preto (para uso em caracteres brancos). Queremos o último, e é por isso que usamos MOV BH,7. Depois, colocamos 0 em AL, o número de linhas a rolar, o que diz a esta rotina para limpar toda a janela, ao invés de subir as linhas.

Agora precisamos modificar nossa rotina de teste, LE_SETOR, para chamar LIMPA_TELA antes de começar a imprimir o setor. Não vamos colocar este CALL em INIC_DISP_SET, pois vamos usar INIC_DISP_SET para reimprimir somente o setor, sem alterar o restante da tela.

Para modificar LE_SETOR, acrescente uma declaração para LIMPA_TELA e insira o CALL LIMPA_TELA. Faça as mudanças a seguir no arquivo ES_DISCO.ASM:

Listagem 17-2. Mudanças em LE_SETOR, de ES_DISCO.ASM

```
        EXTRN   INIC_DISP_SET:NEAR, LIMPA_TELA:NEAR

;-------------------------------------------------------------------------;
; Esta rotina le o primeiro setor do disco A e copia a                    ;
; primeira metade deste setor.                                            ;
;-------------------------------------------------------------------------;
LE_SETOR        PROC    NEAR
        MOV     AL,0
        MOV     CX,1
        MOV     DX,0
        LEA     BX,SETOR
        INT     25h
        POPF
        XOR     DX,DX
        CALL    CLEAR_SCREEN
        CALL    INIC_DISP_SET
        INT     20h
LE_SETOR        ENDP
```

Antes de executar a nova versão de Es_disco, verifique onde o cursor está localizado. Depois, execute Es_disco. A tela se apagará, e Es_disco começará a imprimir no local onde estava o cursor antes de você executar o programa — provavelmente no final da tela.

Mesmo limpando a tela, não falamos nada sobre a posição do cursor no alto da tela. Em BASIC, o comando CLS limpa a tela em duas etapas: ele limpa a tela, e depois move o cursor para o canto superior esquerdo. Nossa rotina não faz isso; temos que mover o cursor por nós mesmos.

Movendo o Cursor

A função 2 de INT 10h seta a posição do cursor de forma parecida com o comando LOCATE do BASIC. Podemos usar GOTO_XY para mover o cursor para qualquer lugar na tela (para o alto, por exemplo, depois de limpar a tela), mas não faremos assim. Entre com esta procedure no arquivo CURSOR.ASM:

Listagem 17-3. Acrescente esta Procedure em CURSOR.ASM

```
        PUBLIC  GOTO_XY

;-------------------------------------------------------------------------
; Esta rotina move o cursor.
;
;     DH      Linha (Y)
;     DL      Coluna (X)
;-------------------------------------------------------------------------
GOTO_XY         PROC    NEAR
        PUSH    AX
        PUSH    BX
        MOV     BH,0            ;Pagina de video 0
        MOV     AH,2            para SETAR POSICAO DO CURSOR
        INT     10h
        POP     BX
        POP     AX
        RET
GOTO_XY         ENDP
```

Usaremos GOTO_XY em uma versão revisada de INIC_DISP_SET, para mover o cursor para a segunda linha antes de apresentarmos o setor. Eis as modificações que você deve fazer:

Listagem 17-4. Alterações em INIC_DISP_SET, de DISP_SET.ASM

```
        PUBLIC  INIC_DISP_SETOR
        EXTRN   IMP_FORMA:NEAR, ENVIA_CRLF:NEAR
        EXTRN   GOTO_XY:NEAR

;-------------------------------------------------------------------------
; Esta rotina inicializa a apresentacao do setor.
;
; Usa:      IMP_FORMA, ENVIA_CRLF, DISP_MEIO_SETOR
;           IMP_INDICES, GOTO_XY
; Le:       FORMA_LIN_SUP, FORMA_LIN_INF
;-------------------------------------------------------------------------
INIC_DISP_SET   PROC    NEAR
        PUSH    DX
        XOR     DL,DL                   ;Move cursor para posicao inicial
        MOV     DH,2                    ;da terceira linha
        CALL    GOTO_XY
          CALL      IMP_INDICES
          LEA       DX,FORMA_LIN_SUP
                .
                .
                .
```

Se você executar o programa agora, verá que a apresentação do setor está bem centralizada.

Como você pode ver, é fácil trabalhar com a tela quando se tem as rotinas do ROM BIOS. No capítulo seguinte, usaremos outra rotina do ROM BIOS para melhorar IMP_CAR, para que imprima qualquer caractere na tela. Mas antes de continuarmos, vamos fazer outras mudanças em nosso

programa, e depois fechar com uma procedure chamada IMP_CAB, que imprimirá a linha de status no alto da tela, mostrando o drive de disco atual e o número do setor.

Ampliando o Uso de Variáveis

Temos muito o que revisar antes de criarmos IMP_CAB. Como elas estão agora, muitas rotinas possuem números fixos em seus códigos; LE_SETOR, por exemplo, lê o setor 0 do drive A. Queremos colocar o drive e o número do setor em variáveis da memória, para que mais de uma rotina possam ler seus valores.

Vamos ter que alterar essas rotinas para que usem variáveis da memória, mas vamos começar colocando todas as variáveis em um só arquivo, DSKPATCH.ASM, para tornar nosso trabalho mais simples. Dskpatch.asm será o primeiro arquivo em nosso programa Dskpatch, de modo que as variáveis serão encontradas com mais facilidade nesse arquivo. Veja agora o arquivo DSKPATCH.ASM, completo com uma longa lista de variáveis da memória:

Listagem 17-5. O Novo Arquivo DSKPATCH.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        ORG     100h

        EXTRN   LIMPA_TELA:NEAR, LE_SETOR:NEAR
        EXTRN   INIC_DISP_SET:NEAR
DISK_PATCH      PROC    NEAR
        CALL    LIMPA_TELA
        CALL    LE_SETOR
        CALL    INIC_DISP_SET
        INT     20h
DISK_PATCH      ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC

        PUBLIC  DESL_SETOR

;-------------------------------------------------------------------------;
; DESL_SETOR e o deslocamento da metade                                   ;
; do setor dentro desse setor. Deve ser                                   ;
; múltiplo de 16 e nao maior que 256.                                     ;
;-------------------------------------------------------------------------;
DESL_SETOR      DW      0

        PUBLIC  NR_SETOR, NR_DRIVE
NR_SETOR        DW      0               ;Inicia setor 0
NR_DRIVE        DB      0               ;Inicia drive A:

        PUBLIC  LINHAS_ANTES_SETOR, NR_LINHA_CAB

        PUBLIC  CAB_1, CAB_2

;-------------------------------------------------------------------------;
; LINHAS_ANTES_SETOR e o numero de linhas                                 ;
; no alto, antes da listagem dos dados                                    ;
; no setor.                                                               ;
;-------------------------------------------------------------------------;
LINHAS_ANTES_SETOR DB   2
NR_LINHA_CAB    DB      0
CAB_1           DB      'Disco ',0
CAB_2           DB      'Setor ',0

        PUBLIC  SETOR
```

```
;-------------------------------------------------------------------------
; O setor inteiro (ate 8192 bytes) esta                                  ;
; armazenado nesta parte da memoria.                                     ;
;-------------------------------------------------------------------------
SETOR     DB        8192 DUP (0)
DATA_SET  ENDS

          END       DISK_PATCH
```

A rotina principal, DISK_PATCH, chama três outras rotinas. Já as vimos antes; logo iremos alterar LE_SETOR e INIC_DISP_SET, para que usem as variáveis colocadas no segmento de dados por esta rotina principal.

Antes que possamos usar Dskpatch, precisamos modificar Disp_set, para trocar a definição de SETOR por EXTRN. Também precisamos alterar Es_disco, para tornar LE_SETOR em uma rotina normal que possa ser chamada a partir de Dskpatch.

Primeiro vamos trabalhar com SETOR. Como já o colocamos em DSKPATCH.ASM, precisamos trocar a definição de SETOR em Disp_set por uma declaração EXTRN. Faça essas alterações em DISP_SET.ASM:

Listagem 17-6. Alterações em DISP_SET.ASM

```
DATA_SEG        SEGMENT PUBLIC
          EXTRN SETOR:BYTE
          PUBLIC SETOR
SETOR     DB    512 DUP(0)

FORMA_LIN_SUP       LABEL     BYTE
          DB        ;?
          DB        SUP_ESQ,1
```

Vamos reescrever o arquivo ES_DISCO.ASM de modo que contenha somente rotinas, e que LE_SETOR use as variáveis da memória (e não números fixos) para os números do setor e do drive. Eis a nova versão de ES_DISCO.ASM:

Listagem 17-7. Alterações em ES_DISCO.ASM

```
CGROUP    GROUP     CODE_SEG, DATA_SEG
          ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG            SEGMENT PUBLIC
          ORG       100h
          PUBLIC    LE_SETOR
DATA_SEG            SEGMENT PUBLIC
          EXTRN     SETOR:BYTE
          EXTRN     NR_DRIVE:BYTE
          EXTRN     NR_SETOR:WORD
DATA_SEG            ENDS
          EXTRN     INIC_DISP_SET:NEAR, LIMPA_TELA:NEAR

;-------------------------------------------------------------------------
; Esta rotina le um setor (512 bytes) para SETOR.                        ;
;                                                                        ;
; Le:       NR_SETOR, NR_DRIVE                                           ;
; Escreve:  SETOR                                                        ;
;-------------------------------------------------------------------------
LE_SETOR            PROC      NEAR
          PUSH      AX
          PUSH      BX
          PUSH      CX
          PUSH      DX
          MOV       AL,NR_DRIVE     ;Numero drive
          MOV       CX,1            ;Le somente 1 setor
          MOV       DX,NR_SETOR     ;Numero do setor logico
```

```
        LEA     BX,SETOR        ;Onde colocar esse setor
        INT     25h             ;Le o setor
        POPF                    ;Descarta os flags colocados na pilha pelo DOS
        XOR     DX,DX           ;Deslocamento 0 dentro de SETOR
        CALL    LIMPA_TELA
        CALL    INIC_DISP_SET   ;Copia a primeira metade
        INT     20h
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LE_SETOR        ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
        EXTRN   SETOR:BYTE
DATA_SEG        ENDS
        END
```

Essa nova versão de Es_disco usa as memórias variáveis NR_DRIVE e NR_SETOR para indicar os números do drive e setor a ser lido. Como essas variáveis já estão definidas em DSKPATCH.ASM, não teremos que alterar Es_disco, quando começarmos a ler setores diferentes de outros drives de disco.

Se você estiver usando o programa Make para reconstruir DSKPATCH.COM, precisará fazer alguns acréscimos ao seu arquivo Make chamando Dskpatch:

Listagem 17-8. A Nova Versão de DSKPATCH

```
dskpach.obj:    dskpatch.asm
        mas dskpatch;

es_disco.obj:   es_disco.asm
        masm es_disco;

disp_set.obj:   disp_set.asm
        masm disp_set;

es_video.obj:   es_video.asm
        masm es_video;

cursor.obj:     cursor.asm
        masm cursor;

dskpach.com:    dskpatch.obj es_disco.obj disp_set.obj es_video.obj cursor.obj
        link diskpatch es_disco disp_set es_video cursor;
        exe2bin diskpatch dskpatch.com
```

Se você não está usando o Make, assegure-se de montar todos os arquivos alterados (Dskpatch, Es_disco e Disp_set), e linkedite todos os cinco arquivos, com Dskpatch listado na frente:

```
LINK DSKPATCH ES_DISCO DISP_SET ES_VIDEO CURSOR;
EXE2BIN DSKPATCH DSKPATCH.COM
```

Fizemos tantas mudanças aqui que será necessário testar o programa Dskpatch para ver se está tudo correndo bem antes de prosseguirmos.

Imprimindo o Cabeçalho

Agora que já convertemos os números do código inicial em variáveis da memória, podemos escrever a rotina IMP_CAB para imprimir uma linha de status no alto da tela. Nosso cabeçalho será assim:

```
Disco A       Setor 0
```

IMP_CAB usará IMP_DECIMAL para imprimir o número do setor corrente em decimal. Essa rotina também imprimirá dois strings de caracteres, *Disco* e *Setor* (cada um seguido por um espaço em branco) e uma letra para o drive (A, por exemplo). Colocaremos a rotina no arquivo ES_VIDEO.ASM.

Para começar, como teremos referências ao segmento de dados (DATA_SEG), altere a primeira linha de ES_VIDEO.ASM (o comando GROUP) para:

```
CGROUP    GROUP       CODE_SEG, DATA_SEG
```

Coloque a seguinte rotina em ES_VIDEO.ASM:

Listagem 17-9. Acrescente esta Rotina em ES_VIDEO.ASM

```
          PUBLIC   IMP_CAB
DATA_SEG  SEGMENT PUBLIC
          EXTRN    NR_LINHA_CAB:BYTE
          EXTRN    CAB_1:BYTE
          EXTRN    CAB_2:BYTE
          EXTRN    NR_DRIVE:BYTE
          EXTRN    NR_SET:WORD
DATA_SEG  ENDS
          EXTRN    GOTO_XY:NEAR

;-------------------------------------------------------------------------;
; Esta rotina imprime o cabeçalho com drive e numero do setor             ;
;                                                                         ;
; Usa:      GOTO_XY, IMP_STRING, IMP_CAR, IMP_DECIMAL                     ;
; Le:       NR_LINHA_CAB, CAB_1, CAB_2                                    ;
;           NR_DRIVE, NR_SETOR                                            ;
;-------------------------------------------------------------------------;
IMP_CAB   PROC     NEAR
          PUSH     DX
          XOR      DL,DL              ;Move o cursor para numero de linha do cab.
          MOV      NR_LINHA_CAB
          CALL     GOTO_XY
          LEA      DX,CAB_1
          CALL     IMP_STRING
          MOV      NR_DRIVE
          ADD      DL,'A'             ;Imprime drives A, B, ...
          CALL     IMP_CAR
          LEA      CAB_2
          CALL     IMP_STRING
          MOV      NR_SETOR
          CALL     IMP_DECIMAL
          POP      DX
          RET
IMP_CAB   ENDP
```

A rotina IMP_STRING ainda não existe. Como você pode ver, planejamos usá-la para imprimir um string de caracteres na tela. Os dois strings, CAB_1 e CAB_2, já foram definidos em DSKPATCH.ASM. IMP_STRING usará DS:DX como o endereço do string.

Escolhemos fazer a nossa própria rotina de impressão de string para que nossos strings pudessem conter qualquer caractere, incluindo o $, que não poderíamos incluir com a função 9 do DOS. Enquanto o DOS usa um $ para marcar o fim do string, usaremos um 0 hexa. Eis a rotina. Entre com ela em ES_VIDEO.ASM:

Listagem 17-10. Acrescente esta Rotina em ES_VIDEO.ASM

```
        PUBLIC      IMP_STRING

;-------------------------------------------------------------------------
; Esta rotina imprime um string de caracteres na tela.
; O string deve terminar com: DB      0
;
;       DS:BX       Endereço do string
;
; Usa:              IMP_CAR
;-------------------------------------------------------------------------
IMP_STRING      PROC    NEAR
        PUSH    AX
        PUSH    DX
        PUSH    SI
        PUSHF                       ;Salva flag direcao
        CLD                         ;Seta direcao p/incremento
        MOV     SI,DX               ;Coloca endereço em SI para LODSB
LOOP_STRING:
        LODSB                       ;Coloca caractere em AL
        OR      AL,AL               ;Ja encontrou o 0?
        JZ      FIM_STRING          ;Sim, terminou o string
        MOV     DL,AL               ;Nao, imprime caractere
        CALL    IMP_CAR
        JMP     LOOP_STRING
FIM_STRING:
        POPF                        ;Restaura flag de direcao
        POP     SI
        POP     DX
        POP     AX
        RET
IMP_STRING      ENDP
```

Conforme está agora, IMP_STRING imprimirá caracteres com códigos ASCII abaixo de 32 (o caractere do espaço) representados por um ponto (.), pois não temos uma versão de IMP_CAR que imprima *qualquer* caractere. Cuidaremos disso em detalhes no próximo capítulo.

Após todo o trabalho que tivemos neste capítulo, vamos colocar a cobertura no bolo. Altere DISK_PATCH em DSKPATCH.ASM de modo que inclua um CALL para IMP_CAB:

Listagem 17-11. Alterações em DISK_PATCH, de DSKPATCH.ASM

```
        EXTRN   LIMPA_TELA:NEAR, LE_SETOR:NEAR
        EXTRN   INIC_DISP_SET:NEAR, IMP_CAB:NEAR
DISK_PATCH      PROC    NEAR
        CALL    LIMPA_TELA
        CALL    IMP_CAB
        CALL    LE_SETOR
        CALL    INIC_DISP_SET
        INT     20h
DISK_PATCH      ENDP
```

Dskpatch deverá agora produzir uma tela semelhante à da Figura 17-1.

Resumo

Finalmente, encontramos as rotinas do ROM BIOS dentro do seu PC, e já usamos duas dessas rotinas para nos ajudar para o nosso alvo, que é o programa Dskpatch completo.

Primeiro, aprendemos sobre INT 10h, função número 6, que usamos para limpar a tela. Também vimos (embora rapidamente) que essa função possui muitos outros usos que não iremos tomar vantagem neste livro. Por exemplo, você pode achar útil descartar periodicamente algumas partes da tela — no Dskpatch ou em seus próprios programas.

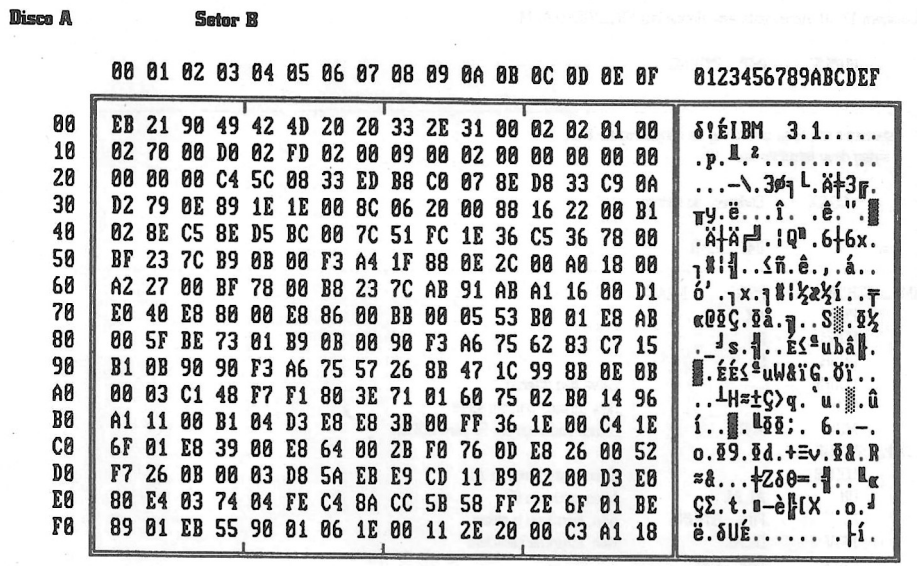


Figura 17-1. ***Dskpatch com o Cabeçalho no Alto.***

Depois usamos a função 2 de INT para mover o cursor até a terceira linha na tela (linha número 2), onde começamos a imprimir o nosso setor.

Para tornar nosso programa bem mais fácil de se trabalhar, alteramos algumas rotinas para que possam utilizar variáveis no lugar de números constantes enraizados no código. Agora, podemos ler outros setores e alterar a forma com que nosso programa trabalha, alterando simplesmente alguns números centrais em DSKPATCH.ASM.

Finalmente, escrevemos as rotinas IMP_CAB e IMP_STRING, para que pudéssemos imprimir um cabeçalho no alto da tela. Como já dissemos, vamos escrever uma versão melhorada de IMP_CAR no próximo capítulo, para que os pontos da janela ASCII possam ser trocados por caracteres gráficos. E, graças ao projeto modular, faremos isso sem alterar qualquer uma das rotinas que usa IMP_CAR.

18

O IMP__CAR DEFINITIVO

Um Novo IMP__CAR

Limpando Até o Fim da Linha

Resumo

Fizemos um bom uso das rotinas do ROM BIOS no último capítulo para limpar a tela e mover o cursor. Mas o ROM BIOS possui muitas outras utilidades, e veremos algumas neste capítulo.

Usando somente o DOS, não podemos apresentar todos os 256 caracteres do IBM PC. Assim, neste capítulo nós vamos fazer com que IMP_CAR apresente qualquer um desses caracteres, graças a uma outra função do ROM_BIOS, e mais especificamente, uma função do VIDEO_IO.

Depois, juntaremos uma outra rotina útil, chamada LIMPA_ATE_FINAL_LINHA, que limpa a linha a partir do cursor até o canto direito. Colocaremos essa rotina em IMP_CAB, para que apague o restante da linha.

Suponha que passemos do setor 10 (dois dígitos) para o setor 9 (um dígito). Seria deixado um 0 do setor 10 quando a rotina fosse imprimir o 9. LIMPA_ATE_FINAL_LINHA apagará esse zero, bem como qualquer outra coisa que estiver no restante da linha.

Um Novo IMP_CAR

A função 9 do ROM BIOS para INT 10h imprime um caractere e seu *atributo* na posição corrente do cursor. O atributo controla características como sublinhado, piscante, cor (veja a descrição das cores em seu Manual de BASIC sob o comando COLOR). Usaremos somente dois atributos em Dskpatch: o atributo 7, que é o atributo normal, e o atributo 70h, que é relativo ao vídeo inverso, com a cor da frente 0 e o fundo 7 (caracteres escuros em fundo claro). Podemos definir os atributos individualmente para cada caractere, e faremos isso depois, para criar um bloco de cursor em vídeo inverso — conhecido como cursor *fantasma*. Por enquanto, entretanto, usaremos somente o atributo normal, quando escrevermos um caractere.:.

INT 10h, função 9, imprime um caractere e o atributo na posição corrente do cursor. Diferente do DOS, ele não avança uma posição de caractere, a menos que você escreva mais de uma cópia do caractere. Usaremos esse fato depois, em uma rotina diferente, mas agora só queremos uma cópia de cada caractere, de modo que moveremos o cursor por nós mesmos.

Eis a nova versão de IMP_CAR, que imprime um caractere e depois move o cursor à direita um caractere. Entre com ele no arquivo ES_VIDEO.ASM:

Listagem 18-1. Mudanças em IMP_CAR, de ES_VIDEO.ASM

```
        PUBLIC    IMP_CAR
        EXTRN     CURSOR_DIREITA:NEAR


;-----------------------------------------------------------------------
; Esta rotina envia um caractere para a tela usando as rotinas do
; ROM BIOS, para que os caracteres especiais sejam testados e impressos
; como outro qualquer.
; Esta rotina deve realizar um certo trabalho, para mudar o cursor.
;
;   DL      Byte a imprimir na tela
;
;  Usa:     CURSOR_DIREITA
;-----------------------------------------------------------------------

IMP_CAR         PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,9                    ;Chamada para sair caractere/atributo
        MOV     BH,0                    ;Define pagina 0
```

```
        MOV     CX,1                ;Imprime somente 1 caractere
        MOV     AL,DL               ;Caractere a imprimir
        MOV     BL,7                ;Atributo normal
        INT     10h                 ;Imprime caractere e atributo
        CALL    CURSOR_DIREITA      ;Move agora a proxima posicao do cursor
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
IMP_CAR ENDP
```

Lendo essa rotina, você pode perguntar-se por que incluímos uma instrução MOV BH,0. Se você possui um adaptador gráfico de vídeo, seu adaptador possui quatro páginas no modo texto normal. Usaremos apenas a primeira página, página 0; por isso, a instrução é importante.

Quanto ao cursor, IMP_CAR usa a rotina CURSOR_DIREITA para mover o cursor uma posição à direita, ou até o início da próxima linha, se for ultrapassada a coluna 79. Coloque a seguinte rotina em CURSOR.ASM:

Listagem 18-2. Acrescenta esta Rotina em CURSOR.ASM

```
        PUBLIC  CURSOR_DIREITA

;-------------------------------------------------------------------------;
; Esta rotina move o cursor uma posicao a direita, ou para a              ;
; proxima linha se estiver no final de uma linha.                         ;
;                                                                         ;
; Usa:      ENVIA_CRLF                                                    ;
;-------------------------------------------------------------------------;
CURSOR_DIREITA  PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3            ;Le a posicao atual do cursor
        MOV     BH,0            ;Pagina ativa = 0
        INT     10h             ;Le posicao do cursor
        MOV     AH,2            ;Define nova posicao
        INC     DL              ;Define coluna seguinte
        CMP     DL,79           ;Coluna deve ser <= 79
        JBE     OK
        CALL    ENVIA_CRLF      ;Vai para linha seguinte
        JMP     FIM
OK:     INT     10h
FIM     POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
CURSOR_DIREITA  ENDP
```

CURSOR_DIREITA usa duas novas funções de INT 10h. A função 3 lê a posição do cursor, e a função 2 altera a posição do cursor. A rotina usa inicialmente a função 3 para descobrir a posição atual do cursor, que é retornado em dois bytes — o número da coluna em DL, e o número da linha em DH. Depois, a rotina incrementa o número da coluna (em DL) e move o cursor. Se DL indicar a última coluna (79), a rotina envia um par mudança de linha/retorno de carro para mover o cursor até a próxima linha. Não precisamos verificar essa coluna 79 em nosso programa Dskpatch, mas fazendo isso, nossa rotina CURSOR_DIREITA torna-se uma procedure de uso geral, que você pode usar em qualquer um de seus programas.

Com estas mudanças, Dskpatch deve, agora, apresentar todos os 256 caracteres, como você pode ver pela Figura 18-1.

Você pode verificar que ele realmente cumpriu o prometido, procurando um caractere com valor menor que 20h e comparando a janela da direita. Algum caractere estranho deve aparecer no lugar do ponto inicialmente apresentado.

Agora vamos fazer algo talvez ainda mais interessante: vamos escrever uma rotina para limpar uma linha da posição do cursor até o final.

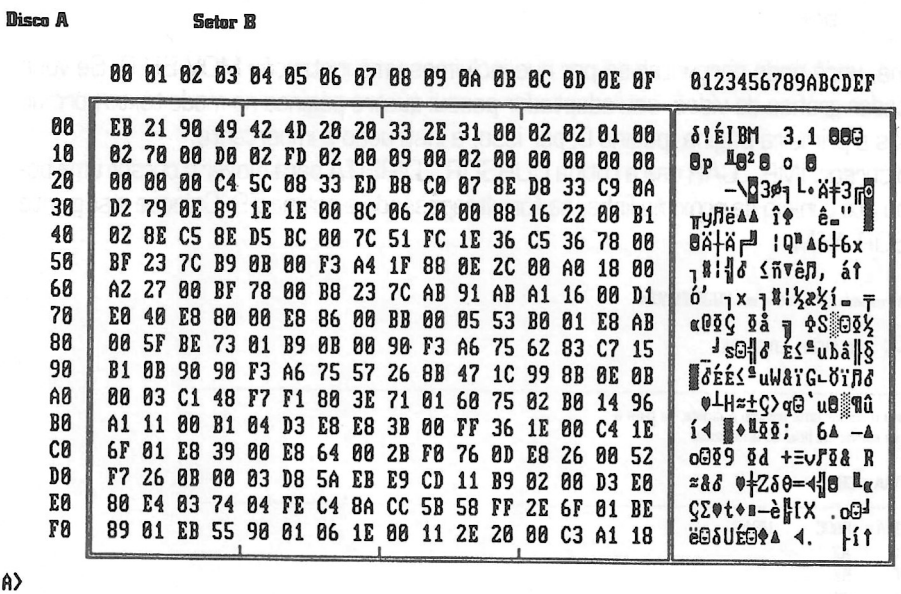


Figura 18-1. ***Dskpatch com o Novo IMP_CAR.***

Limpando Até o Fim da Linha

No capítulo anterior, usamos a função 6 de INT 10h para limpar a tela na rotina LIMPA_TELA. Naquela ocasião, dissemos que a função 6 poderia limpar qualquer janela retangular dentro da tela física. Essa capacidade aplica-se também a janelas com uma linha de altura e menos de uma linha de largura, de modo que podemos usar a função 6 para limpar parte da linha — até o fim da linha.

O lado esquerdo da janela, nesse caso, é o número de coluna da posição do cursor, que obtemos com uma chamada da função 3 (também usada por CURSOR_DIREITA). O lado direito da janela é sempre a coluna 79. Você pode ver os detalhes em LIMPA_ATE_FINAL_LINHA; coloque a rotina em CURSOR.ASM:

Listagem 18-3. Acrescente esta rotina em CURSOR.ASM

```
        PUBLIC  LIMPA_ATE_FINAL_LINHA

;-----------------------------------------------------------------------
; Esta rotina limpa a linha da posicao atual do cursor
; ate o final da linha.
;-----------------------------------------------------------------------
LIMPA_ATE_FINAL_LINHA   PROC    NEAR
        PUSH    AX
        PUSH    BX
```

```
        PUSH    CX
        PUSH    DX
        MOV     AH,3            ;Le a posicao atual do cursor
        XOR     BH,BH           ;na pagina 0
        INT     10h             ;Agora temos (X,Y) em DL,DH
        MOV     AH,6            ;Define limpar ate fim da linha
        XOR     AL,AL           ;Limpa janela
        MOV     CH,DH           ;Tudo na mesma linha
        MOV     CL,DL           ;Comeca na posicao do cursor
        MOV     DL,79           ;e para ao final da linha
        MOV     BH,7            ;Usa atributo normal
        INT     10h
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LIMPA_ATE_FINAL_LINHA   ENDP
```

Usaremos essa rotina em IMP_CAB, para limparmos o restante da linha quando começarmos a ler outros setores (o que não vai demorar muito). Não há um modo de você verificar o funcionamento de LIMPA_ATE_FINAL_LINHA juntamente com IMP_CAB antes de incluirmos as novas rotinas que irão ler um setor e atualizar o setor, mas vamos revisar IMP_CAB de uma vez, e depois não precisaremos voltar a alterá-lo.

Listagem 18-4. Alterações em IMP_CAB, de ES_VIDEO.ASM

```
        PUBLIC  IMP_CAB
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_LINHA_CAB:BYTE
        EXTRN   CAB_1:BYTE
        EXTRN   CAB_2:BYTE
        EXTRN   NR_DRIVE:BYTE
        EXTRN   NR_SETOR:WORD
DATA_SEG        ENDS
        EXTRN   GOTO_XY:NEAR, LIMPA_ATE_FINAL_LINHA:NEAR
;-------------------------------------------------------------------------
; Esta rotina imprime o cabecalho com drive e numero do setor.
;
; Usa:    GOTO_XY, IMP_STRING, IMP_CAR, IMP_DECIMAL
;         LIMPA_ATE_FINAL_LINHA
; Le:     NR_LINHA_CAB, CAB_1, CAB_2
;         NR_DRIVE, NR_SETOR
;-------------------------------------------------------------------------
IMP_CAB         PROC    NEAR
        PUSH    DX
        XOR     DL,DL                   ;Move cursor para numero de linha do cab.
        MOV     NR_LINHA_CAB
        CALL    GOTO_XY
        LEA     DX,CAB_1
        CALL    IMP_STRING
        MOV     DL,NR_DRIVE
        ADD     DL,'A'                  Imprime drives A, B, ...
        CALL    IMP_CAR
        LEA     DX,CAB_2
        CALL    IMP_STRING
        MOV     DX,NR_SETOR
        CALL    IMP_DECIMAL
        CALL    LIMPA_ATE_FINAL_LINHA;Limpa restante do numero do setor
        POP     DX
        RET
IMP_CAB         ENDP
```

Esta revisão marca tanto a versão final de IMP_CAB, quanto o término de CURSOR.ASM. Ainda estão faltando algumas partes muito importantes em Dskpatch (não se preocupe, ainda teremos muito trabalho). No próximo capítulo, vamos prosseguir incluindo o controlador principal para os comandos do teclado, e poderemos pressionar F1 e F2 para ler outros setores do disco.

Resumo

Este capítulo foi relativamente fácil, sem muita informação nova ou truques. Aprendemos a usar a função 9 de INT 10h, no ROM BIOS, para escrever qualquer caractere na tela.

Com isso, vimos também como podemos ler a posição do cursor com a função 3 da mesma interrupção, para que pudéssemos mover o cursor em uma posição à direita após escrevermos um caractere. A razão: a função 9 de INT 10h não move o cursor depois da impressão de um caractere, a não ser que façamos mais de uma cópia desse caractere. Finalmente, colocamos em ação a função 6, para limpar parte de uma mesma linha.

No próximo capítulo, vamos novamente voltar ao assunto montando o controlador central.

19

O CONTROLADOR

O Controlador

Lendo Outros Setores

Filosofia dos Capítulos Seguintes

É bom, em qualquer linguagem, termos um programa bem escrito que faça algo, mas para realmente trazer um programa à vida é necessário torná-lo interativo. É da natureza humana dizer: "Se eu fizer isso, você faz aquilo", de modo que usaremos este capítulo para incluir alguma interação humana a Dskpatch. Escreveremos uma rotina para entrada no teclado e um controlador central. A tarefa do controlador será chamar a rotina correta para cada tecla pressionada. Por exemplo, quando você pressionar F1 para ler e apresentar o setor anterior, o controlador chamará uma rotina chamada ANT_SETOR. Para isso, faremos muitas mudanças em Dskpatch. Começaremos criando CONTROLA, o controlador central, e algumas outras rotinas para formatação de vídeo. A seguir, colocaremos duas rotinas, ANT_SETOR e PROX_SETOR, que chamaremos através de CONTROLA.

O Controlador

CONTROLA será o controle central de Dskpatch, de modo que toda a entrada e edição serão feitas por meio dele. A tarefa de CONTROLA será ler os caracteres e chamar outras rotinas para fazer o trabalho. Você logo verá como o controlador faz o seu trabalho, mas primeiro vamos ver como ele se encaixa em Dskpatch.

CONTROLA terá sua própria linha de pedidos (ou pronto), logo abaixo da apresentação do setor, onde aguardará uma entrada pelo teclado. Você não poderá entrar com números hexa em nossa primeira versão da rotina de entrada do teclado, mas depois poderá. Veja agora as primeiras modificações em DSKPATCH.ASM; elas acrescentam os dados para a linha de pedido:

Listagem 19-1. Acréscimos em DATA_SEG, de DSKPATCH.ASM

```
HEADER_LINE_NO      DB      0
HEADER_PART_1       DB      'Disco',0
HEADER_PART_2       DB      '       Setor ',0
        PUBLIC
NR_LINHA_PRONTO     DB      21
EDITOR_PRONTO       DB      'Press tecla funcao, ou entre
                    DB      'caractere ou byte hexa: ',0
```

Colocaremos outras mensagens depois para cuidar de assuntos como a entrada de um número para um novo setor, de modo que nossa tarefa ficará mais simples usando uma rotina comum, IMP_PRONTO, para imprimir cada linha de pronto. Cada rotina que usar IMP_PRONTO fornecerá o endereço da mensagem (aqui, o endereço de EDITOR_PRONTO), e depois ela será impressa na linha 21 (pois NR_LINHA_PRONTO é 21). Por exemplo, essa nova versão de DISK_PATCH (em DSKPATCH.ASM) usa IMP_PRONTO logo antes de chamar CONTROLA:

Listagem 19-2. Acréscimos em DISK_PATCH, de DSKPATCH_ASM

```
        EXTRN   LIMPA_TELA:NEAR, LE_SETOR:NEAR
        EXTRN   INIC_DISP_SET:NEAR, IMP_CAB:NEAR
        EXTRN
                IMP_PRONTO:NEAR, CONTROLE:NEAR
DISK_PATCH      PROC    NEAR
        CALL    LIMPA_TELA
        CALL    IMP_CAB
        CALL    LE_SETOR
        CALL    INIC_DISP_SET
        LEA     EDITOR_PRONTO
        CALL    IMP_PRONTO
        CALL    CONTROLA
        INT     20h
DISK_PATCH      ENDP
```

O controlador em si é um programa muito simples, mas usamos alguns truques novos nele. A listagem a seguir é a nossa primeira versão do arquivo CONTROLE.ASM:

Listagem 19-3. O Novo Arquivo CONTROLE.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  CONTROLA
        EXTRN   LE_BYTE:NEAR


;-------------------------------------------------------------------------------
; Este e o controlador central. Durante a edicao e apresentacao normal,       ;
; esta rotina le caracteres do teclado e, se for uma tecla de                  ;
; comando (como tecla de cursor), CONTROLA chama as rotinas                    ;
; que fazem o trabalho real. Este controle e feito para as telas               ;
; especiais listadas em TAB_CONTROLE, onde os endereços das                    ;
; rotinas sao colocados apos os nomes das teclas.                              ;
;    Se o caractere nao for especial, entao deve ser colocado                  ;
; diretamente no buffer do setor -- este e o modo de edicao.                   ;
;                                                                              ;
; Usa:    LE_BYTE                                                              ;
;-------------------------------------------------------------------------------

CONTROLA        PROC    NEAR
        PUSH    AX
        PUSH    BX
LOOP_CONT:
        CALL    LE_BYTE         ;Le caractere em AX
        OR      AH,AH           ;AX = 0 se nenhum caractere, -1
                                ;para codigo estendido
        JZ      LOOP_CONT       ;Nenhum caractere, tenta de novo
        JS      TECLA_ESPECIAL  ;Le codigo estendido
                                ;nao faz nada com o caractere por enquanto
        JMP     LOOP_CONT       ;Le outro caractere
TECLA_ESPECIAL:
        CMP     AL,68           ;F10 — saida?
        JE      FIM_CONTROLE    ;Sim, sai
                                ;Usa BX para verificar tabela
        LEA     BX,TAB_CONTROLE
LOOP_ESPECIAL:
        CMP     BYTE PTR [BX],0 ;Fim da tabela?
        JE      FORA_TABELA     ;Sim, tecla nao estava na tabela
        CMP     AL,[BX]         ;Achou o codigo entrado?
        JE      CONTROLE        ;Sim, envia
        ADD     BX,3            ;Nao, tenta a proxima entrada
        JMP     LOOP_ESPECIAL   ;Verifica proxima entrada na tabela

CONTROLE:
        INC     BX              ;Aponta endereco da rotina
        CALL    WORD PTR [BX]   ;Chama rotina
        JMP     LOOP_CONT       ;Espera outra tecla

FORA:TABELA:                    ;Somente le o proximo caractere
        JMP     LOOP_CONT

FIM_CONTROLE:
        POP     BX
        POP     AX
        RET
CONTROLA        ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC

CODE_SEG        SEGMENT PUBLIC
        EXTRN   PROX_SETPR:NEAR ;Em ES_DISCO.ASM
        EXTRN   ANT_SETOR:NEAR  ;Em ES_DISCO.ASM
CODE_SEG        ENDS
```

```
;-----------------------------------------------------------------------;
; Esta tabela contem as teclas ASCII estendidas validas e os enderecos  ;
; das rotinas que devem ser chamadas quando a tecla for pressionada.    ;
;    O formato da tabela e:                                             ;
;         DB      60          Codigo estendido para F2                  ;
;         DW      OFFSET CGROUP:PROX_SETOR                              ;
;-----------------------------------------------------------------------;

TAB_CONTROLE    LABEL     BYTE
        DB      59                      ;F1
        DW      OFFSET CGROUP:ANT_SETOR
        DB      60                      ;F2
        DW      OFFSET CGROUP:PROX_SETOR
        DB      0                       ;Fim da tabela
DATA_SEG        ENDS
```

TAB_CONTROLE contém o código ASCII estendido das teclas F1 e F2. Cada código é seguido do endereço da rotina que o programa deve chamar quando lê um código estendido em particular. Por exemplo, quando LE_BYTE, que é chamado por CONTROLA, lê uma tecla F1 (código estendido 59), CONTROLA chama a rotina ANT_SETOR.

O endereço das rotinas que estamos controlando estão na tabela, e elas são obtidas por meio da pseudo-op OFFSET. A linha

```
DW      OFFSET CGROUP:ANT_SET
```

por exemplo, diz ao assembler para usar o *deslocamento* (*offset*) da rotina ANT_SET. Este deslocamento é calculado em relação ao início do nosso grupo CGROUP, e é por isso que precisamos do "CGROUP:" logo antes do nome da rotina. Se não tivéssemos colocado o CGROUP lá, o assembler calcularia o endereço da rotina a partir do início do segmento de código, e isso poderia não ser o que queremos. (Aqui, no entanto, esse CGROUP não é absolutamente necessário, pois o segmento de código é carregado em primeiro lugar no nosso programa. Mesmo assim, por questão de clareza, iremos escrever OFFSET CGROUP: antes do nome da rotina.)

Note que TAB_CONTROLE possui tanto bytes quanto palavras. Isso merece alguma explicação. Anteriormente, só usamos tabelas de um tipo ou de outro: só com palavras (2 bytes), ou só com bytes. Mas aqui estamos usando ambos, de modo que temos que informar ao assembler qual o tipo do dado que vamos usar com o CALL ou CMP. No caso de uma instrução escrita da seguinte forma:

```
CMP     [BX],0
```

o assembler não sabe se queremos comparar duas palavras ou dois bytes. Mas, se escrevermos a instrução dessa forma:

```
CMP     BYTE PTR [BX],0
```

dizemos ao assembler que BX aponta para um byte, e que queremos uma comparação de byte. Da mesma forma, a instrução CMP WORD PTR [BX],0 compararia palavras. Por outro lado, uma instrução como CMP,AL[BX] não causa qualquer problema, pois AL é um registrador de byte, e o assembler sabe sem qualquer aviso que a comparação deve ser feita com bytes.

Lembre-se, também, que a instrução CALL pode ser um CALL NEAR ou FAR. Se for NEAR, ele precisa apenas de uma palavra como endereço, enquanto um CALL FAR precisa de duas. Aqui, a instrução:

```
CALL    WORD PTR [BX]
```

diz ao assembler que [BX] aponta para uma palavra, de modo que geraria um CALL curto e usaria a palavra apontada por [BX] como endereço, como o endereço que armazenamos em TAB_CONTROLE. (Para um CALL longo, que usa endereço de duas palavras, usaríamos a instrução CALL DWORD PTR [BX]. DWORD significa *Dupla Palavra*.

Como você verá no Capítulo 22, podemos facilmente acrescentar novos comandos de tecla a Dskpatch, acrescentando mais rotinas e colocando novas entradas em TAB_CONTROLE. Agora, no entanto, precisamos acrescentar quatro rotinas antes que possamos testar nossa nova versão de Dskpatch. Está faltando ainda LE_BYTE, IMP_PRONTO, ANT_SETOR e PROX_SETOR.

LE_BYTE é uma rotina que lê caracteres e códigos ASCII estendidos do teclado. A versão final poderá ler caracteres especiais (como as teclas de função e de movimento do cursor), caracteres ASCII, e números hexa de dois dígitos. Neste ponto, escreveremos uma versão simplificada de LE_BYTE — para ler ou um caractere ou uma tecla especial. Eis a primeira versão de ES_TEC.ASM, que é o arquivo onde colocaremos nossas rotinas que lêem o teclado.

Listagem 19-4. O Novo Arquivo ES_TEC.ASM

```
CGROUP  GROUP   CODE_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        PUBLIC  LE_BYTE


;-----------------------------------------------------------------------
; Esta rotina le um unico caractere ASCII. Ela e somente uma
; versao de teste de LE_BYTE.
;
; Retorna byte em       AL      Codigo do caractere (o mesmo que AH = 0)
;                       AH      1 se leu um caractere ASCII
;                               -1 se leu uma tecla especial
;-----------------------------------------------------------------------
LE_BYTE         PROC    NEAR
        MOV     AH,?            ;Le caractere sem imprimir
        INT     21h             ;e o coloca em AL
        OR      AL,AL           ;E um codigo estendido?
        JZ      COD_ESTENDIDO   ;Sim
NAO_ESTENDIDO:
        MOV     AH,1            ;Sinaliza caractere ASCII normal
FIM_LEITURA:
        RET

COD_ESTENDIDO:
        INT     21h             ;Le o codigo estendido
        MOV     AH,0FFh         ;Sinaliza codigo estendido
        JMP     FIM_LEITURA
LE_BYTE         ENDP

CODE_SEG        ENDS

        END
```

Colocaremos IMP_PRONTO em ES_VIDEO, da seguinte forma:

Listagem 19-5. Acrescente esta Rotina em ES_VIDEO.ASM

```
        PUBLIC  IMP_PRONTO
        EXTRN   LIMPA_ATE_FINAL_LINHA:NEAR
        EXTRN   GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_LINHA_PRONTO:BYTE
DATA_SEG        ENDS

;-----------------------------------------------------------------------
; Esta rotina imprime a linha de pronto na tela e limpa ate o
; fim da linha
;
;       DS:DX       Endereço da mensagem
;
; Usa:              IMP_STRING, LIMPA_ATE_FINAL_LINHA, GOTO_XY
; Le:               NR_LINHA_PRONTO
;-----------------------------------------------------------------------
```

```
IMP_PRONTO      PROC    NEAR
        PUSH    DX
        XOR     DL,DL                   ;Imprime a linha de pronto e
        MOV     DH,NR_LINHA_PRONTO      ;move o cursor
        CALL    GOTO_XY
        POP     DX
        CALL    IMP_STRING
        CALL    LIMPA_ATE_FINAL_LINHA
        RET
IMP_PRONTO              ENDP
```

Na verdade, não há muita coisa nessa rotina. Ela move o cursor para o início da linha de pronto, que foi definida como 21 (em DSKPATCH.ASM). Depois, ela imprime uma linha de pronto e limpa o restante da linha. Quando a rotina termina, o cursor estará no final da linha de pronto, e o restante da linha é apagado por LIMPA_ATE_FINAL_LINHA.

Lendo Outros Setores

Finalmente, precisamos de duas rotinas, ANT_SETOR e PROX_SETOR, para ler os setores adjacentes do disco. Acrescente essas duas rotinas no arquivo ES_DISCO.ASM:

Listagem 19-6. Acrescente Estas Rotinas em ES_DISCO.ASM

```
        PUBLIC  ANT_SETOR
        EXTRN   INIC_DISP_SET:NEAR, IMP_CAB:NEAR
        EXTRN   IMP_PRONTO:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_SETOR:WORD, EDITOR_PRONTO:BYTE
DATA_SEG        ENDS

;-----------------------------------------------------------------------;
; Esta rotina le o setor anterior, se possivel.                         ;
;                                                                       ;
; Usa:          IMP_CAB, LE_SETOR, INIC_DISP_SET                        ;
;               IMP_PRONTO                                              ;
; Le:           NR_SETOR, EDITOR_PRONTO                                 ;
; Imprime:      NR_SETOR                                                ;
;-----------------------------------------------------------------------;
ANT_SETOR       PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     AX,NR_SETOR             ;Apanha numero do setor
        OR      AX,AX                   ;Nao decrementa se for 0
        JZ      NAO_DECREMENTA
        DEC     AX
        MOV     NR_SETOR                ;Salva novo numero do setor
        CALL    IMP_CAB
        CALL    LE_SETOR
        CALL    INIC_DISP_SET           ;Apresenta novo setor
        LEA     DX,EDITOR_PRONTO
        CALL    IMP_PRONTO
NAO_DECREMENTA:
        POP     DX
        POP     AX
        RET
ANT_SETOR               ENDP

        PUBLIC  PROX_SETOR
        EXTRN   INIC_DISP_SET:NEAR, IMP_CAB:NEAR
        EXTRN   IMP_PRONTO:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_SETOR:WORD, EDITOR_PRONTO:BYTE
DATA_SEG        ENDS
```

```
;-------------------------------------------------------------------------
; Le o proximo setor.
;
; Usa:          IMP_CAB, LE_SETOR, INIC_DISP_SET
;               IMP_PRONTO
; Le:           NR_SETOR, EDITOR_PRONTO
; Imprime:      NR_SETOR
;-------------------------------------------------------------------------
PROX_SETOR      PROC    NEAR
                PUSH    AX
                PUSH    DX
                MOV     AX,NR_SETOR
                INC     AX                      ;Move para proximo cursor
                MOV     NR_SETOR,AX
                CALL    IMP_CAB
                CALL    LE_SETOR
                CALL    INIC_DISP_SET           ;Apresenta novo setor
                LEA     DX,EDITOR_PRONTO
                CALL    IMP_PRONTO
                POP     DX
                POP     AX
                RET
PROX_SETOR      ENDP
```

Agora você está pronto para montar todos os arquivos que criamos ou alteramos: Dskpatch, Es_video, Es_téc, Controle e Es_disco. Quando você linkeditar os arquivos, lembre-se de que agora há sete deles: Cursor e Disp_set devem ser acrescentados à lista acima.

Se você estiver usando o Make, eis os acréscimos que você deve fazer ao arquivo Dskpatch (a barra invertida na quarta linha de baixo para cima indica que estamos continuando a lista na linha seguinte:

Listagem 19-7. Alterações no Arquivo Make DSKPATCH

```
cursor.obj:     cursor.asm
    masm cursor;

controle.obj:   controle.asm
    masm controle;

es_tec.obj:     es_tec
    masm es_tec;

dskpatch.com:   dskpatch.obj es_disco.obj disp_set.obj es_video.obj cursor.obj \
    controle.obj es_tec.obj
    link dskpatch es_disco disp_set es_video cursor controle es_tec;
    exe2bin dskpatch dskpatch.com
```

Se você não possui o programa Make, você pode escrever o seguinte arquivo de lote para linkeditar e criar seu arquivo .COM:

```
LINK DSKPATCH ES_DISCO DISP_SET ES_VIDEO CURSOR CONTROLE ES_TEC;
EXE2BIN DSKPATCH DSKPATCH.COM
```

Conforme acrescentamos mais arquivos, você terá que alterar esse arquivo de lote, ao invés de digitar essa longa lista de linkedição toda vez que montar o arquivo .COM.

Esta versão de Dskpatch possui três teclas ativas: F1 lê e apresenta o setor anterior, parando no setor 0; F2 lê o setor seguinte; F10 sai do Dskpatch. Tente usar essas teclas. Sua tela deve ficar semelhante à Figura 19-1.

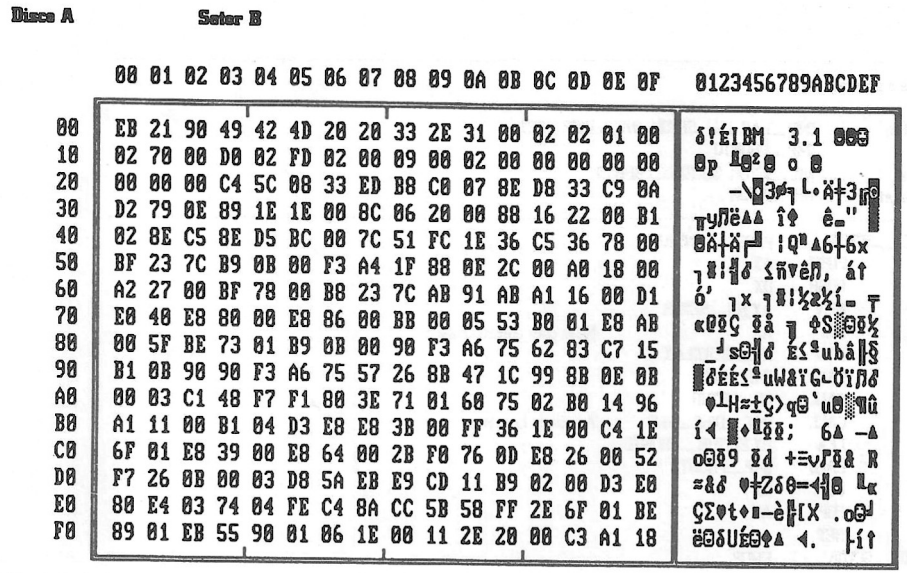


Figura 19-1. ***Dskpatch com a Linha de Pedido.***

Filosofia dos Capítulos Seguintes

Neste capítulo, vimos mais do que o costumeiro, e com isso você deve ter sentido a filosofia que aplicaremos nos Capítulos de 20 a 27. Daqui por diante, vamos seguir num passo bem rápido, para que possamos passar por mais exemplos de como escrever grandes programas. Você também verá muitas rotinas que podem ser usadas em seus próprios programas. Quando você voltar atrás, verá muitas dicas de programação.

Logicamente, se você estiver afoito para escrever suas próprias rotinas, leia o capítulo seguinte. Lá, você encontrará inúmeras dicas, e dar-lhe-emos uma chance de escrever suas próprias rotinas para os capítulos seguintes, fornecendo diversos detalhes para que você prossiga.

Do Capítulo 21 em diante, apresentaremos muitas rotinas diferentes e deixaremos que você descubra como elas funcionam. Por quê? Há duas razões, ambas relacionadas a deixar você caminhar com seus próprios pés e com suas próprias formas de programar em linguagem assembly. Primeiro, queremos que você tenha uma biblioteca de rotinas para usar em seus programas; para utilizá-las confiadamente, você precisa exercitar suas próprias. Segundo, apresentando este grande exemplo de programação, queremos mostrar-lhe não só como escrever um grande programa, mas dar-lhe um sentido para ele.

Assim, use o restante desse livro como achar melhor. O Capítulo 20 é para aqueles que querem escrever logo seus próprios programas. No Capítulo 21, voltaremos ao Dskpatch, para montar as rotinas que imprimem e movem o que chamamos de cursor fantasma: um cursor em vídeo reverso para as apresentações hexa e ASCII na tela.

20

UM DESAFIO DE PROGRAMAÇÃO

Os Cursores Fantasma

Edição Simples

Outros Acréscimos e Mudanças em Dskpatch

Este livro contém mais seis capítulos de rotinas. Se você quiser tentar prosseguir por conta própria, leia este capítulo. Vamos apresentar aqui um guia para você passar pelos Capítulos 21 e 22. Depois você pode tentar escrever as rotinas antes de ler nos capítulos. Se você quiser tentar fazer sozinho as outras partes de Dskpatch, salte este capítulo por enquanto. Ele é muito breve e deixa muitos detalhes para a sua imaginação.

Se você quiser ler este capítulo, eis uma sugestão de como fazer: leia uma seção e depois tente fazer as correções em Dskpatch. Quando você sentir que fez bastante progresso, leia o capítulo com o título correspondente à seção. Depois de ter lido o capítulo correspondente, você prossegue lendo a outra seção.

Nota: Você pode fazer agora mesmo uma cópia de seus arquivos antes de começar as mudanças. Quando você chegar ao Capítulo 21, terá a possibilidade de escolher entre realizar as mudanças ou seguir sua própria versão.

Os Cursores Fantasmas

No Capítulo 21, colocaremos dois cursores fantasmas na tela: um na janela hexa, e um na janela ASCII. Um cursor fantasma é semelhante ao cursor normal, mas ele não pisca, e o fundo torna-se branco, com os caracteres em preto, como você pode ver na Figura 20-1. (Note que uma reprodução em papel sempre inverte a "cor" da tela.)

O cursor fantasma na janela hexa ocupa quatro caracteres, enquanto na janela ASCII ele ocupa somente um.

Como podemos criar o cursor fantasma? Cada caractere na tela possui um byte de *atributo*. Este byte diz ao seu IBM PC como apresentar cada caractere. Um código de atributo 7h apresenta um caractere normal, enquanto 70h apresenta um caractere em vídeo reverso. O último é exata-

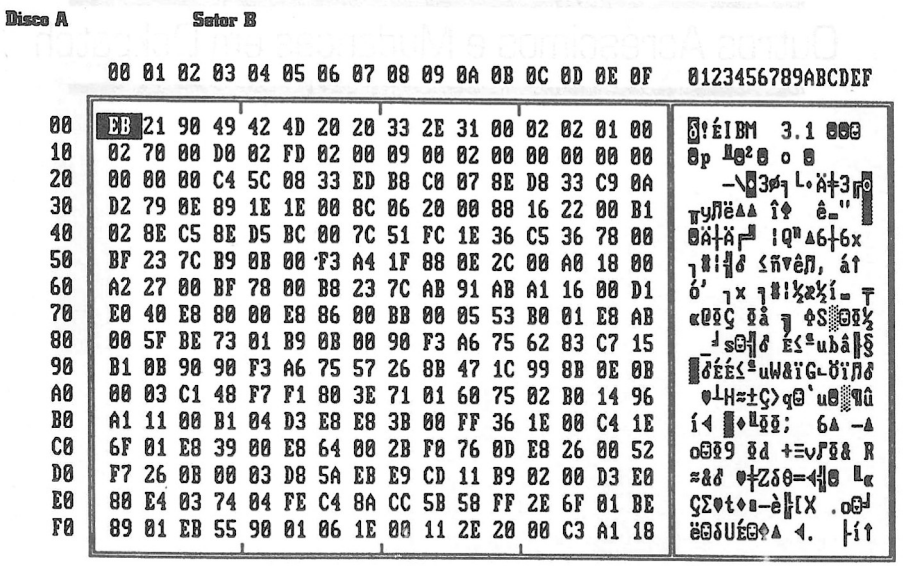


Figura 20-1. ***Uma Tela com Cursores Fantasma.***

mente o que queremos para o nosso cursor fantasma, de modo que a questão é: Como podemos mudar o atributo dos caracteres para 70h?

A função 9 de INT 10h coloca um caractere e o seu atributo na tela, e a função 8 da mesma interrupção lê o caractere na posição corrente do cursor. Podemos criar um cursor fantasma na janela hexa com os seguintes passos:

- Salve a posição do cursor real (use INT 10h, função 3, para ler a posição do cursor, e salve essa posição em uma variável).
- Mova o cursor real para o início do cursor fantasma na janela hexa.
- Para os próximos quatro caracteres, leia o código do caractere (função 8) e imprima tanto o caractere quanto o seu atributo (definindo-o como 70h).
- Finalmente, recupere a antiga posição do cursor.

Imprimimos um cursor na janela ASCII da mesma forma. Quando você tiver o cursor fantasma para a janela hexa, poderá então acrescentar o código extra para a janela ASCII.

Tenha em mente que sua primeira tentativa será apenas temporária. Quando você tiver um programa funcionando com cursores fantasma, pode voltar e escrever suas mudanças, de modo que você tenha uma série de pequenas rotinas para fazer o trabalho. Olhe para as rotinas no Capítulo 21 quando tiver terminado, e veja um modo de se fazer isso.

Edição Simples

Uma vez tendo nossos cursores fantasma, vamos querer movê-los pela tela. Temos que prestar atenção nas condições de limite, a fim de manter os cursores dentro de cada janela. Também queremos que nossos dois cursores se movam juntos, pois eles são duas representações de um mesmo valor.

Como podemos mover cada cursor fantasma? Cada uma das quatro teclas de movimentação de cursor, no seu teclado, envia um número de função especial: 72 para cursor acima, 80 para cursor abaixo, 75 para cursor à esquerda, e 77 para cursor à direita. Esses são os números que precisamos acrescentar em TAB_CONTROLE, junto com o endereço das quatro rotinas para mover os cursores fantasma pelas quatro direções.

Para mover realmente um cursor, apague-o, altere suas duas coordenadas e imprima-o novamente. Se você teve cuidado ao escrever suas rotinas para os cursores fantasma, as quatro rotinas para movê-los serão muito fáceis.

Sempre que você digitar um caractere no teclado, Dskpatch deve ler este caractere e trocar o byte sob o cursor fantasma pelo caractere lido. Eis os passos para uma simples edição:

- Leia o caractere do teclado.
- Altere o número em hexa na janela hexa e o caractere na janela ASCII para que combine com o caractere lido.
- Altere o byte no buffer do setor, SETOR.

Eis uma dica: você não precisa fazer muitas mudanças para adicionar a edição. Dispatch requer pouco mais do que a chamada a uma nova rotina (nós a chamamos de EDITA_BYTE) que realiza quase todo o trabalho. EDITA_BYTE é responsável pela mudança na tela e em SETOR.

Outros Acréscimos e Mudanças em Dskpatch

Do Capítulo 23 ao 27, as mudanças passam a usar mais truques. Se você estiver realmente interessado em escrever sua própria versão, pense nisso: o que mais você gostaria de ter em Dskpatch além do que já tem? Usamos as idéias a seguir nos outros capítulos.

Queremos uma nova versão de LE_BYTE que leia ou um caractere ou um número hexa de dois dígitos e espere que pressionemos a tecla Enter antes que retorne o caractere para Dskpatch. Essa parte da nossa "lista de desejos" não é tão simples quanto parece, e gastaremos dois capítulos (Capítulos 23 e 24) trabalhando em cima disso.

No Capítulo 25, vamos caçar erros, e no Capítulo 26 vamos aprender a retornar os setores modificados ao disco, usando a função INT 26h do DOS, que é semelhante à INT 25h que usamos para ler o setor do disco. (No Capítulo 26, não vamos checar os erros de leitura, mas você verá como isso pode ser feito na versão final de Dskpatch, que se encontra ao final deste livro.)

Finalmente, no Capítulo 27, faremos algumas mudanças em Dskpatch para que você possa ver a outra metade do nosso setor. Estas mudanças não permitem que você liste o setor "linha a linha" livremente conforme queríamos, mas essas mudanças encontram-se na versão em disco do Dskpatch.

21

OS CURSORES FANTASMA

Os Cursores Fantasma

Alterando Atributos de Caracteres

Resumo

Neste capítulo, vamos escrever as rotinas para colocar a retirar o cursor fantasma nas janelas hexa e ASCII. Um cursor fantasma é chamado assim, porque ele não é o cursor de hardware do PC; é uma sombra ... embora uma sombra um tanto incomum, pois ela inverte o caractere, tornando o fundo claro e o caractere escuro. Na janela hexa, temos espaço para marcar esse caractere com quatro colunas de largura, para que possa ser visto com mais facilidade. Na janela ASCII, nosso cursor fantasma terá apenas um caractere de largura, pois não há espaço entre os caracteres.

Temos diversas rotinas e códigos para ver aqui, de modo que vamos descrevê-las resumidamente.

Os Cursores Fantasma

INIC_DISP_SET é a única rotina que temos para alterar a apresentação do setor. Uma nova tela aparece quando iniciamos o Dskpatch, e toda vez que lemos um novo setor. Como nossos cursores estarão na apresentação do setor, começaremos nosso trabalho colocando uma chamada para IMP_FANTASMA em INIC_DISP_SET. Dessa forma, teremos um cursor fantasma toda vez que colocarmos um novo setor na tela.

Veja agora a versão revisada — e final — de INIC_DISP_SET em DISP_SET.ASM:

Listagem 21-1. Alterações em INIC_DISP_SET, de DISP_SET.ASM

```
            PUBLIC   INIC_DISP_SET
            EXTRN    IMP_FORMA:NEAR, ENVIA_CRLF:NEAR
            EXTRN    GOTO_XY:NEAR, IMP_FANTASMA:NEAR
DATA_SEG    SEGMENT PUBLIC
            EXTRN    DESL_SETOR:WORD
DATA_SEG    ENDS
;-------------------------------------------------------------------------
; Esta rotina inicializa a apresentacao do setor.                        ;
;                                                                        ;
; Usa:        IMP_FORMA, ENVIA_CRLF, DISP_MEIO_SETOR                     ;
;             IMP_INDICES, GOTO_XY, IMP_FANTASMA                         ;
; Le:         FORMA_LIN_SUP, FORMA_LIN_INF                               ;
; Escreve:    DESL_SETOR                                                 ;
;                                                                        ;
;-------------------------------------------------------------------------
INIC_DISP_SET   PROC    NEAR
            PUSH    DX
            XOR     DL,DL                   ;Move cursor para posicao inicial
            MOV     DH,LINHAS_ANTES_SETOR
            CALL    GOTO_XY
            CALL    IMP_INDICES
            LEA     DX,FORMA_LIN_SUP
            CALL    IMP_FORMA
            CALL    ENVIA_CRLF
            XOR     DX,DX               ;Começa no inicio do setor
            MOV     DESL_SETOR          ;Deslocamento do setor = 0
            CALL    DISP_MEIO_SETOR
            LEA     FORMA_LIN_INF
            CALL    IMP_FORMA
            CALL    IMP_FANTASMA        ;Imprime o cursor fantasma
            POP     DX
            RET
INIC_DISP_SET   ENDP
```

Veja bem que nós atualizamos INIC_SET_DISP para que use e inicialize variáveis. Agora a rotina zera o DESL_SETOR, para apresentar a primeira metade do setor.

Vamos agora continuar com o próprio IMP_FANTASMA. Isso vai levar um pouco de tempo e trabalho. Ao todo, temos que escrever seis rotinas, incluindo IMP_FANTASMA. A idéia, porém, é muito simples. Primeiro, movemos o cursor real para a posição do cursor fantasma na janela hexa e alteramos o atributo dos quatro caracteres seguintes (atributo 70h). Isto cria um bloco em branco,

com quatro caracteres de largura, e com os números hexa em "preto". Depois, fazemos o mesmo com a janela ASCII, mas para um único caractere. Finalmente, movemos o cursor real de volta para onde estava quando começamos. Todas essas rotinas para o cursor fantasma estarão no arquivo FANTASMA.ASM, com exceção de IMP_ATRIBUTO_N_VEZES, a rotina que mudará o atributo dos caracteres.

Entre com as rotinas a seguir no arquivo FANTASMA.ASM:

Listagem 21-2. O Novo Arquivo FANTASMA.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        PUBLIC  GOTO_XY:NEAR

DATA_SEG        SEGMENT PUBLIC
        EXTRN   LINHAS_ANTES_SETOR
DATA_SEG        ENDS

;-----------------------------------------------------------------------------;
; Esta rotina move o cursor real para a posicao do cursor                     ;
; fantasma na janela hexa.                                                    ;
;                                                                             ;
; Usa:      GOTO_XY                                                           ;
; Le:       LINHAS_ANTES_SETOR, CURSOR_FANT_X, CURSOR_FANT_Y                  ;
;-----------------------------------------------------------------------------;

MOVE_POS_HEXA PROC      NEAR
        PUSH    AX
        PUSH    CX
        PUSH    DX
        MOV     DH,LINHAS_ANTES_SETOR   ;Encontra linha do fantasma (0,0)
        ADD     DH,2                    ;Mais linha de hexa e barra horizontal
        ADD     DH,CURSOR_FANT_Y        ;DH = linha do cursor
        MOV     DL,8                    ;Indenta a esquerda
        MOV     CL,3                    ;Cada coluna usa 3 caracteres,
        MOV     AL,CURSOR_FANT_X        ;e devemos multiplicar CURSOR_X por 3
        MUL     CL
        ADD     DL,AL                   ;e somar a indentacao
        CALL    GOTO_XY                 ;para obter a coluna do cursor fantasma
        POP     DX
        POP     CX
        POP     AX
        RET
MOVE_POS_HEXA ENDP

        PUBLIC  MOVE_POS_ASCII
        EXTRN   GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   LINHAS_ANTES_SETOR:BYTE
DATA_SEG        ENDS

;-----------------------------------------------------------------------------;
; Esta rotina move o cursor real para o inicio do cursor                      ;
; fantasma na janela ASCII.                                                   ;
;                                                                             ;
; Usa:      GOTO_XY                                                           ;
; Le:       LINHAS_ANTES_SETOR, CURSOR_FANT_X, CURSOR_FANT_Y                  ;
;-----------------------------------------------------------------------------;

MOV_POS_ASCII   PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     DH,LINHAS_ANTES_SETOR   ;Encontra linha do fantasma (0,0)
        ADD     DH,2                    ;Mais linha seguinte
        ADD     DH,CURSOR_FANT_Y        ;DH = linha do cursor fantasma
        MOV     DL,59                   ;Indenta a esquerda
        ADD     DL,CURSOR_FANT_X        ;Soma CURSOR_X para obter
```

```
        CALL    GOTO_XY                 ;a posicao do cursor fantasma
        POP     DX
        POP     AX
        RET
MOVE_POS_ASCII      ENDP

        PUBLIC  SALVA_CURSOR_REAL

;-------------------------------------------------------------------------
; Esta rotina salva a posicao do cursor real
; nas duas variaveis abaixo:
;
; Escreve:      CURSOR_REAL_X, CURSOR_REAL_Y
;-------------------------------------------------------------------------
SALVA_CURSOR_REAL   PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Le posicao do cursor
        XOR     BH,BH                   ;na pagina 0
        INT     10h                     ;E a retorna em DL,DH
        MOV     CURSOR_REAL_Y,DL        ;Salva posicao
        MOV     CURSOR_REAL_X,DH
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
SALVA_CURSOR_REAL   ENDP

        PUBLIC  RESTAURA_CURSOR_REAL
        EXTRN   GOTO_XY:NEAR

;-------------------------------------------------------------------------
; Esta rotina restaura o cursor real a sua antiga posicao,
; salva pelas variáveis abaixo.
;
; Usa:      GOTO_XY
; Le:       CURSOR_REAL_X, CURSOR_REAL_Y
;-------------------------------------------------------------------------

RESTAURA_CURSOR_REAL    PROC    NEAR
        PUSH    DX
        MOV     DL,CURSOR_REAL_Y
        MOV     DH,CURSOR_REAL_X
        CALL    GOTO_XY
        POP     DX
        RET
RESTAURA_CURSOR_REAL    ENDP

        PUBLIC  IMP_FANTASMA
        EXTRN   IMP_ATRIBUTO_N_VEZES:NEAR

;-------------------------------------------------------------------------
; Esta rotina usa CURSOR_X e CURSOR_Y, atraves de MOVE_POS_...,
; como coordenadas para o cursor fantasma. IMP_FANTASMA imprime
; o cursor fantasma.
;
; Usa:      IMP_ATRIBUTO_N_VEZES, SALVA_CURSOR_REAL
;           RESTAURA_CURSOR_REAL, MOVE_POS_HEXA
;           MOVE_POS_ASCII
;-------------------------------------------------------------------------
IMP_FANTASMA    PROC    NEAR
        PUSH    CX
        PUSH    DX
        CALL    SALVA_CURSOR_REAL
        CALL    MOV_POS_HEXA            ;Coord. do cursor na janela hexa
```

```
        MOV     CX,4                    ;Cursor possui 4 caracteres de largura
        MOV     DL,70h
        CALL    IMP_ATRIBUTO_N_VEZES
        CALL    MOVE_POS_ASCII          ;Coord. do cursor na janela ASCII
        MOV     CX,1                    ;Cursor possui 1 caractere de largura
        CALL    IMP_ATRIBUTO_N_VEZES
        CALL    RESTAURA_CURSOR_REAL
        POP     DX
        POP     CX
        RET
IMP_FANTASMA    ENDP
        PUBLIC  APAGA_FANTASMA
        EXTRN   IMP_ATRIBUTO_N_VEZES:NEAR

;-------------------------------------------------------------------------
; Esta rotina apaga o cursor fantasma, que e o oposto de
; IMP_FANTASMA.
;
; Usa:    IMP_ATRIBUTO_N_VEZES, SALVA_CURSOR_REAL
;         RESTAURA_CURSOR_REAL, MOVE_POS_HEXA
;         MOVE_POS_ASCII
;-------------------------------------------------------------------------

APAGA_FANTASMA PROC     NEAR
        PUSH    CX
        PUSH    DX
        CALL    SALVA_CURSOR_REAL
        CALL    MOVE_POS_HEXA           ;Coordenada do cursor na janela hexa
        MOV     CX,4                    ;Muda de volta para branco no fundo preto
        MOV     DL,7
        CALL    IMP_ATRIBUTO_N_VEZES
        CALL    MOVE_POS_ASCII
        MOV     CX,1
        CALL    IMP_ATRIBUTO_N_VEZES
        CALL    RESTAURA_CURSOR_REAL
        POP     DX
        POP     CX
        RET
APAGA_FANTASMA ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
CURSOR_REAL_X           DB      0
CURSOR_REAL_Y           DB      0
        PUBLIC  CURSOR_FANT_X, CURSOR_FANT_Y
CURSOR_REAL_X           DB      0
CURSOR_REAL_Y           DB      0
DATA_SEG        ENDS

        END
```

IMP_FANTASMA e APAGA_FANTASMA são quase a mesma coisa. Na verdade, a única diferença está no atributo utilizado: IMP_FANTASMA emite um atributo 70h para o vídeo reverso, enquanto APAGA_FANTASMA coloca o atributo de volta ao normal (7h).

As duas rotinas guardam a posição do cursor real com SALVA_CURSOR_REAL, que usa a função 3 de INT 10h para ler a posição atual do cursor e depois salva essa posição nos dois bytes CURSOR_REAL_X e CURSOR_REAL_Y.

Após salvar a posição do cursor real, as duas rotinas IMP_FANTASMA e APAGA_FANTASMA chamam a rotina MOVE_POS_HEXA, que move o cursor para o início do cursor fantasma na janela hexa. A seguir, IMP_ATRIBUTO_N_VEZES coloca o atributo de vídeo reverso para os quatro caracteres, começando no cursor, e movendo-se para a direita. Isso coloca o cursor na janela hexa. Da mesma forma, IMP_FANTASMA coloca um cursor com largura de um caractere

na janela ASCII. Finalmente, RESTAURA_CURSOR_REAL restaura o cursor real à posição que ele ocupava antes da chamada de IMP_FANTASMA. (Lembre-se: o cursor real não é um cursor físico, como o do BASIC ! Ele é apenas o local onde será feita qualquer entrada no setor, expresso em coordenada X e Y.)

A única rotina que ficou faltando foi IMP_ATRIBUTO_N_VEZES, e vamos cuidar dela agora.

Alterando Atributos de Caracteres

Vamos usar IMP_ATRIBUTO_N_VEZES para fazer três coisas. Primeiro, ele vai ler o caractere sob a posição do cursor. Precisamos disso porque a função 9 de INT 10h, que iremos usar para setar o atributo de um caractere, imprime *tanto* o caractere *quanto* o seu atributo na posição do cursor. Assim, essa rotina mudará o atributo imprimindo o novo atributo juntamente com o caractere lido. Finalmente, a rotina moverá o cursor para a direita até a próxima posição, para que possamos repetir esse processo inteiro N vezes (quatro, no caso da janela hexa). Você poderá ver os detalhes na própria rotina; coloque IMP_ATRIBUTO_N_VEZES no arquivo ES_VIDEO.ASM:

Listagem 21-3. Acrescente Esta Rotina em ES_VIDEO.ASM

```
        PUBLIC  IMP_ATRIBUTO_N_VEZES
        EXTRN   CURSOR_DIREITA:NEAR

;-----------------------------------------------------------------------
; Esta rotina coloca o atributo em N caracteres, começando
; na posicao atual do cursor.
;
;       CX      Numero de caracteres colocar o atributo
;       DL      Novo atributo dos caracteres
;
; Usa:          CURSOR_DIREITA
;-----------------------------------------------------------------------

IMP_ATRIBUTO_N_VEZES    PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BL,DL           ;Seta novo atributo
        XOR     BH,BH           ;Pagina de video = 0
        MOV     DX,CX           ;CX e usado pelo ROM BIOS
        MOV     CX,1            ;Seta atributo para um caractere
LOOP_ATR:
        MOV     AH,8            ;Le o caractere sob o cursor
        INT     10h
        MOV     AH,9            ;Imprime atributo/caractere
        INT     10h
        CALL    CURSOR_DIREITA
        DEC     DX              ;Ja imprimiu N caracteres?
        JNZ     LOOP_ATR        ;Nao, continua
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
IMP_ATRIBUTO_N_VEZES    ENDP
```

Esta é a primeira e definitiva versão de IMP_ATRIBUTO_N_VEZES. Com isso, terminamos a versão final de ES_VIDEO.ASM, de modo que você não precisará mais alterá-la ou montá-la daqui para a frente.

```
Disco A          Setor B

     00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
00 | EB 21 90 49 42 4D 20 20 33 2E 31 00 02 02 01 00 |
10 | 02 70 00 D0 02 FD 02 00 09 00 02 00 00 00 00 00 |
20 | 00 00 00 C4 5C 08 33 ED B8 C0 07 8E D8 33 C9 0A |
30 | D2 79 0E 89 1E 1E 00 8C 06 20 00 88 16 22 00 B1 |
40 | 02 8E C5 8E D5 BC 00 7C 51 FC 1E 36 C5 36 78 00 |
50 | BF 23 7C B9 0B 00 F3 A4 1F 88 0E 2C 00 A0 18 00 |
60 | A2 27 00 BF 78 00 B8 23 7C AB 91 AB A1 16 00 D1 |
70 | E0 40 E8 80 00 E8 86 00 BB 00 05 53 B0 01 E8 AB |
80 | 00 5F BE 73 01 B9 0B 00 90 F3 A6 75 62 83 C7 15 |
90 | B1 0B 90 90 F3 A6 75 57 26 8B 47 1C 99 8B 0E 0B |
A0 | 00 03 C1 48 F7 F1 80 3E 71 01 60 75 02 B0 14 96 |
B0 | A1 11 00 B1 04 D3 E8 E8 3B 00 FF 36 1E 00 C4 1E |
C0 | 6F 01 E8 39 00 E8 64 00 2B F0 76 0D E8 26 00 52 |
D0 | F7 26 0B 00 03 D8 5A EB E9 CD 11 B9 02 00 D3 E0 |
E0 | 80 E4 03 74 04 FE C4 8A CC 5B 58 FF 2E 6F 01 BE |
F0 | 89 01 EB 55 90 01 06 1E 00 11 2E 20 00 C3 A1 18 |

Press tecla funcao, ou entre caractere ou byte hexa:
```

Figura 21-1. ***Apresentação na Tela com Cursores Fantasma.***

Resumo

Agora já temos oito arquivos para linkeditar, com a rotina principal ainda em Dskpatch. Desses oito, mudamos dois arquivos, Disp_set e Es_video, e criamos um, Fantasma. Se você estiver usando Make ou um pequeno arquivo de lote igual ao que sugerimos no Capítulo 20, lembre-se de acrescentar seu novo arquivo, Fantasma, à lista.

Quando você executar Dskpatch agora, verá a apresentação do mesmo setor, porém com os dois cursores fantasma. (Veja a Figura 21-1.) Note que o cursor está onde realmente deveria estar, no canto superior.

No próximo capítulo, colocaremos rotinas para mover nossos cursores fantasma recém-formados, e ainda uma simples rotina que nos permita alterar o byte sob o cursor fantasma.

22

EDIÇÃO SIMPLES

Movendo os Cursores Fantasma

Edição Simples

Resumo

Estamos quase chegando ao ponto onde poderemos editar nosso setor na tela — alterar os valores da metade do setor. Logo acrescentaremos versões simples das rotinas para editar bytes em nosso vídeo, mas antes disso precisamos, de algum modo, mover os cursores fantasma para bytes diferentes dentro do meio setor. Esta tarefa torna-se muito mais fácil do que poderia ser, pois agora já temos as duas rotinas IMP_FANTASMA e APAGA_FANTASMA.

Movendo os Cursores Fantasma

O movimento dos cursores para qualquer direção depende de três passos básicos: apagar o cursor de sua posição atual; alterar a posição do cursor, modificando uma das variáveis CURSOR_FANT_X ou CURSOR_FANT_Y; e usar IMP_FANTASMA para imprimir o cursor fantasma na nova posição. Nesse processo, entretanto, devemos evitar que o cursor ultrapasse os limites da janela, que possui 16 bytes de largura e 16 bytes de altura.

Para mover o cursor, precisamos de quatro novas rotinas, uma para cada tecla de movimentação de cursor. A rotina CONTROLA não precisa de mudanças, pois toda a informação sobre as rotinas e os códigos estendidos está na tabela TAB_CONTROLE. Só precisamos acrescentar os códigos ASCII estendidos e os endereços das rotinas para cada uma das teclas de seta. Veja, a seguir, os acréscimos necessários em CONTROLE.ASM, que trarão as teclas de cursor à vida:

Listagem 22-1. Alterações em CONTROLE.ASM

```
DATA_SEG        SEGMENT PUBLIC
CODE_SEG        SEGMENT PUBLIC
        EXTRN   PROX_SETOR:NEAR                         ;Em ES_DISCO.ASM
        EXTRN   ANT_SETOR:NEAR                          ;Em ES_DISCO.ASM
        EXTRN   FANTASMA_CIMA:NEAR, FANTASMA_BAIXO:NEAR ;Em FANTASMA.ASM
        EXTRN   FANTASMA_ESQ:NEAR, FANTASMA_DIR:NEAR
CODE_SEG        ENDS

;-----------------------------------------------------------------------
; Esta tabela contem as teclas ASCII estendidas validas e os enderecos
; das rotinas que devem ser chamadas quando a tecla for pressionada.
;   O formato da tabela e:
;       DB      60          Codigo estendido para F2
;       DW      OFFSET CGROUP:PROX_SETOR
;-----------------------------------------------------------------------

TAB_CONTROLE    LABEL   BYTE
        DB      59                                      ;F1
        DW      OFFSET CGROUP:ANT_SETOR
        DB      60                                      ;F2
        DW      OFFSET CGROUP:PROX_SETOR
        DB      72                                      ;Seta para cima
        DW      OFFSET CGROUP:FANTASMA_CIMA
        DB      80                                      ;Seta para baixo
        DW      OFFSET CGROUP:FANTASMA_BAIXO
        DB      75                                      ;Seta esquerda
        DW      OFFSET CGROUP:FANTASMA_ESQ
        DB      77                                      ;Seta direita
        DW      OFFSET CGROUP:FANTASMA_DIR
        DB      0
DATA_SEG        ENDS
```

Como você pode ver, é simples acrescentar comandos a Dskpatch: simplesmente colocamos os nomes das rotinas em TAB_CONTROLE e escrevemos as rotinas.

E por falar em rotinas, as rotinas a seguir são muito simples, e são todas semelhantes às outras, diferindo apenas nas condições de limite de cada uma. Já descrevemos como elas trabalham; veja se você pode escrevê-las por si mesmo, no arquivo FANTASMA.ASM, antes de lê-las.

Aqui estão nossas versões para as rotinas que movem os cursores fantasma:

Listagem 22-2. Acrescente Estas Rotinas em FANTASMA.ASM

```
;-----------------------------------------------------------------------
; Estas quatro rotinas movem os cursores fantasma.
;
; Usa:      APAGA_FANTASMA, IMP_FANTASMA
; Le:       CURSOR_FANT_X, CURSOR_FANT_Y
; Altera:   CURSOR_FANT_X, CURSOR_FANT_Y
;-----------------------------------------------------------------------

            PUBLIC  FANTASMA_CIMA
FANTASMA_CIMA   PROC    NEAR
            CALL    APAGA_FANTASMA  ;Apaga na posicao atual
            DEC     CURSOR_FANT_Y   ;Sobe cursor uma linha
            JNS     NAO_ESTAVA_TOPO ;Nao estava no alto, imprime
            MOV     CURSOR_FANT_Y,0 ;Estava no alto, coloca de volta
NAO_ESTAVA_TOPO:
            CALL    IMP_FANTASMA    ;Imprime o cursor na nova posicao
            RET
FANTASMA_CIMA   ENDP

            PUBLIC  FANTASMA_BAIXO
FANTASMA_BAIXO  PROC    NEAR
            CALL    APAGA_FANTASMA  ;Apaga na posicao atual
            INC     CURSOR_FANT_Y   ;Abaixa cursor uma linha
            CMP     CURSOR_FANT_Y,16 ;Estava no fundo?
            JB      NAO_ESTAVA_FUNDO;Nao, imprime fantasma
            MOV     CURSOR_FANT_Y,15 ;Sim, coloca de volta
NAO_ESTAVA_FUNDO:
            CALL    IMP_FANTASMA    ;Imprime o cursor fantasma
            RET
FANTASMA_BAIXO  ENDP

            PUBLIC  FANTASMA_ESQ
FANTASMA_ESQ    PROC    NEAR
            CALL    APAGA_FANTASMA  ;Apaga na posicao atual
            DEC     CURSOR_FANT_X   ;Move uma coluna a esquerda
            JNS     NAO_ESTAVA_ESQ  ;Nao estava do lado direito, imprime cursor
            MOV     CURSOR_FANT_X,0 ;Estava na esquerda, coloca de volta
NAO_ESTAVA_ESQ:
            CALL    IMP_FANTASMA    ;Imprime o cursor fantasma
            RET
FANTASMA_ESQ    ENDP

            PUBLIC  FANTASMA_DIR
FANTASMA_DIR    PROC    NEAR
            CALL    APAGA_FANTASMA  ;Apaga na posicao atual
            INC     CURSOR_FANT_X   ;Move uma coluna a direita
            CMP     CURSOR_FANT_X,16 ;Ja esta no lado direito?
            JB      NAO_ESTAVA_DIR
            MOV     CURSOR_FANT_X,15 ;Estava na direita, coloca de volta
NAO_ESTAVA_DIR:
            CALL    IMP_FANTASMA    ;Imprime o cursor fantasma
            RET
FANTASMA_DIR    ENDP
```

FANTASMA_ESQ e FANTASMA_DIR são as versões finais, mas teremos que alterar FANTASMA_CIMA e FANTASMA_BAIXO quando começarmos a mover a tela.

Como Dskpatch está agora, podemos ver somente a primeira metade de um setor. No Capítulo 27, veremos alguns acréscimos e mudanças para que ele possa mover a tela e ver outras partes do setor. Nessa ocasião, vamos alterar FANTASMA_CIMA e FANTASMA_BAIXO para que movam a tela, quando tentarmos mover o cursor além do topo ou da base da tela. Por exemplo, quando o cursor estiver na última linha do meio setor, a tela inteira subiria uma linha, e a linha seguinte seria acrescentada no lugar da antiga última linha, e dessa forma veríamos os próximos 16 bytes. Entretanto essa tarefa de "rolar" a tela é um tanto confusa, e por isso a deixaremos para o fim. Através do

Capítulo 16, vamos desenvolver as seções de edição e entrada pelo teclado de Dskpatch, usando apenas o primeiro meio setor.

Teste Dskpatch para ver se você pode movimentar o cursor livremente pela tela. Eles devem mover-se juntos, permanecendo dentro de suas janelas. Vamos agora continuar acrescentando a edição, para que possamos alterar os bytes em nossa tela.

Edição Simples

Já temos uma rotina simples de entrada pelo teclado, LE_BYTE, que lê somente um caractere do teclado sem esperar que você pressione a tecla Enter. Usaremos essa versão antiga de teste para desenvolvermos nossa edição. Depois, no capítulo seguinte, escreveremos uma versão mais sofisticada da rotina que irá esperar até que pressionemos ou a tecla Enter ou uma tecla especial, como uma tecla de função ou cursor.

Nossa rotina de edição será chamada EDITA_BYTE, e ela vai mudar não só a tela como a memória (SETOR). EDITA_BYTE colocará o caractere no registrador DL, alterará a posição de memória atualmente apontada pelo cursor fantasma, e depois alterará a tela.

CONTROLA já tem um bom local para colocarmos um CALL para EDITA_BYTE. Eis a nova versão de CONTROLA em CONTROLE.ASM, com a chamada para EDITA_BYTE e as alterações necessárias:

Listagem 22-3. Alterações em CONTROLA, de CONTROLE.ASM

```
        PUBLIC    CONTROLA
        EXTRN     LE_BYTE:NEAR, EDITA_BYTE:NEAR

;-----------------------------------------------------------------------;
; Este e o controlador central. Durante a edicao e apresentacao normal,;
; esta rotina le caracteres do teclado e, se for uma tecla de          ;
; comando (como tecla de cursor), CONTROLA chama as rotinas            ;
; que fazem o trabalho real. Este controle e feito para as telas       ;
; especiais listadas em TAB_CONTROLE, onde os endereços das            ;
; rotinas sao colocados apos os nomes das teclas.                      ;
;    Se o caractere nao for especial, entao deve ser colocado          ;
; diretamente no buffer do setor -- este e o modo de edicao.           ;
;                                                                      ;
; Usa:      LE_BYTE, EDITA_BYTE                                        ;
;-----------------------------------------------------------------------;
CONTROLA        PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    DX
LOOP_CONT:
        CALL    LE_BYTE             ;Le caractere em AL
        OR      AH,AH               ;AH = 0 se nenhum caractere,
                                    ; -1 para codigo estendido
        JZ      LOOP_CONT           ;Nenhum caractere, tenta de novo
        JS      TECLA_ESPECIAL      ;Le codigo estendido
Nao faz nada com o caractere por enquanto
        MOV     DL,AL
        CALL    EDITA_BYTE          ;Foi caractere normal, edita byte
        JMP     LOOP_CONT           ;Le outro caractere

TECLA_ESPECIAL:
        CMP     AL,68               ;F10 -- saida?
        JE      FIM_CONTROLE        ;Sim, sai
                                    ;Usa BX para verificar tabela
        LEA     BX,TAB_CONTROLE
LOOP_ESPECIAL:
        CMP     BYTE PTR [BX],0     ;Fim da tabela?
        JE      FORA_TABELA         ;Sim, tecla nao estava na tabela
```

```
            CMP     AL,[BX]             ;Achou o codigo entrado?
            JE      CONTROLE            ;Sim, envia
            ADD     BX,3                ;Nao, tenta proxima entrada
            JMP     LOOP_ESPECIAL       ;Verifica proxima entrada na tabela
CONTROLE:
            INC     BX                  ;Aponta endereco da rotina
            CALL    WORD PTR [BX]       ;Chama rotina
            JMP     LOOP_CONT           ;Espera outra tecla
FORA_TABELA:                            ;Somente le o proximo caractere
            JMP     LOOP_CONT

FIM_CONTROLE:
            POP     DX
            POP     BX
            POP     AX
            RET
CONTROLA    ENDP
```

A rotina EDITA_BYTE por si só não faz muita coisa, e quase só chama outras rotinas. Esta é uma característica do projeto modular. Com o projeto modular, podemos escrever rotinas bem complexas simplesmente dando uma lista de CALLs para outras rotinas que fazem o trabalho real. Muitas das rotinas em EDITA_BYTE trabalham com um caractere no registrador DL, mas isso já está definido quando chamamos EDITA_BYTE, de modo que a única instrução que precisamos além da CALL (ou PUSH, POP) é a instrução LEA para definir o endereço do pronto para IMP_PRONTO. A maioria das chamadas de rotina em EDITA_BYTE são para a atualização do vídeo quando editamos um byte. Você verá os outros detalhes de EDITA_BYTE quando chegarmos à listagem da rotina, logo em seguida.

Como EDITA_BYTE altera o byte na tela, precisamos de outra rotina, ALTERA_MEMORIA, para alterar o byte em SETOR. ALTERA_MEMORIA usa as coordenadas em CURSOR_FANT_X e CURSOR_FANT_Y para calcular o deslocamento em SETOR do cursor fantasma, e depois coloca o caractere indicado por DL no byte correto dentro do setor.

Listagem 22-4. O Novo Arquivo EDITOR.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT   PUBLIC

DATA_SEG        SEGMENT   PUBLIC
        EXTRN   SETOR:BYTE
        EXTRN   DESL_SETOR:WORD
        EXTRN   CURSOR_FANT_X:BYTE
        EXTRN   CURSOR_FANT_Y:BYTE
DATA_SEG        ENDS

;-----------------------------------------------------------------------------;
; Esta rotina coloca um byte em SETOR, para posicao de memoria                ;
; apontado pelo cursor fantasma.                                              ;
;                                                                             ;
;     DL      Byte a ser alocado em SETOR                                     ;
;                                                                             ;
; O deslocamento e calculado por                                              ;
;   DESL = DESL_SETOR + (17 * CURSOR_FANT_Y) + CURSOR_FANT_X                  ;
;                                                                             ;
; Le:     CURSOR_FANT_X, CURSOR_FANT_Y, DESL_SETOR                            ;
; Altera: SETOR                                                               ;
;-----------------------------------------------------------------------------;
ALTERA_MEMORIA  PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DESL_SETOR
```

```
        MOV     AL,CURSOR_FANT_Y
        XOR     AH,AH
        MOV     CL,4                    ;Multiplica CURSOR_FANT_Y por 16
        SHL     AX,CL
        ADD     BX,AX                   ;BX = DESL_SETOR + (16 * Y)
        MOV     AL,CURSOR_FANT_X
        XOR     AH,AH
        ADD     BX,AX                   ;Eis o endereco!
        MOV     SETOR[BX],DL            ;Agora, armazena o byte.
        POP     CX
        POP     BX
        POP     AX
        RET
ALTERA_MEMORIA      ENDP

        PUBLIC  EDITA_BYTE
        EXTRN   SALVA_CURSOR_REAL:NEAR, RESTAURA_CURSOR_REAL:NEAR
        EXTRN   MOVE_POS_HEXA:NEAR, MOVE_POS_ASCII:NEAR
        EXTRN   IMP_FANTASMA:NEAR, IMP_PRONTO:NEAR
        EXTRN   CURSOR_DIREITA:NEAR, IMP_HEXA:NEAR, IMP_CAR:NEAR
DATA_SEG    SEGMENT PUBLIC
        EXTRN   EDITOR_PRONTO:BYTE
DATA_SEG    ENDS

;-------------------------------------------------------------------------------
; Esta rotina troca o byte na memoria e na tela
;
;       DL      Byte a colocar em SETOR, e alterar na tela
;
; Usa:          SALVA_CURSOR_REAL, RESTAURA_CURSOR_REAL
;               MOVE_POS_HEXA, MOVE_POS_ASCII
;               IMP_FANTASMA, IMP_PRONTO, CURSOR_DIREITA
;               IMP_HEXA, IMP_CAR, ALTERA_MEMORIA
; Le:           EDITOR_PRONTO
;-------------------------------------------------------------------------------
EDITA_BYTE      PROC    NEAR
        PUSH    DX
        CALL    SALVA_CURSOR_REAL
        CALL    MOV_POS_HEXA            ;Move para o numero hexa
        CALL    CURSOR_DIREITA          ;   na janela hexa
        CALL    IMP_HEXA                ;Imprime o novo numero
        CALL    MOVE_POS_ASCII          ;Move o caractere na janela ASCII
        CALL    IMP_CAR                 ;Imprime o novo caractere
        CALL    RESTAURA_CURSOR_REAL    ;Move o cursor de volta para onde estava
        CALL    IMP_FANTASMA            ;Reimprime o cursor fantasma
        CALL    ALTERA_MEMORIA          ;Salva este novo byte em SETOR
        LEA     DX,EDITOR_PRONTO
        CALL    IMP_PRONTO
        POP     DX
        RET
EDITA_BYTE      ENDP

CODE_SEG        ENDS

        END
```

Resumo

Dskpatch agora contém nove arquivos: Dskpatch, Controle, Disp_set, Es_disco, Es_video, Es_tec, Fantasma, Cursor e Editor. Neste capítulo, alteramos Controle e acrescentamos Editor. Nenhum desses arquivos é muito grande, de modo que nenhum deles leva muito tempo para ser montado. Além disso, podemos fazer as mudanças muito rapidamente, editando só um desses arquivos, montando-o, e depois linkeditando todos os arquivos novamente.

Em termos de nossa versão atual de Dskpatch, aperte qualquer tecla e você verá uma mudança no número e no caractere sob o cursor fantasma. Nossa versão funciona, mas ainda não é muito segura, pois podemos alterar um byte esbarrando em qualquer tecla. Precisamos montar algum tipo de segurança, como pressionar Enter para alterar um byte, de modo que não seja feita uma mudança no setor, ao pressionarmos acidentalmente uma tecla do teclado.

Além disso, a versão atual de LE_BYTE não nos permite entrar com um número hexa para alterar um byte. No Capítulo 24, iremos reescrever LE_BYTE, para que exija o toque na tecla Enter antes de alterar um caractere e nos permite entrar com um número hexa de dois dígitos. Primeiro, precisamos escrever uma rotina para entrada em hexa; no próximo capítulo, vamos escrever rotinas de entrada em hexa e decimal.

23

ENTRADA HEXA E DECIMAL

Entrada Hexa

Entrada Decimal

Resumo

Neste capítulo, encontraremos duas novas rotinas para entrada pelo teclado: uma rotina para a leitura de um byte ou pelo número hexa de dois dígitos ou pelo caractere, e outra para ler uma palavra por meio dos caracteres de um número decimal. Essas serão nossas rotinas de entrada hexa e decimal.

As duas rotinas utilizam certos truques, de forma que precisamos até mesmo de um programa de teste antes de pensarmos em linkeditá-las ao programa Dskpatch. Trabalharemos com LE_BYTE, uma rotina de teste que será particularmente importante aqui, pois esta rotina perderá (temporariamente) sua capacidade de ler as teclas de função especial. Como o Dskpatch baseia-se na teclas de função, não poderemos usar nosso novo LE_BYTE com o Dskpatch. Também veremos por que não podemos usar as teclas de função especial com o LE_BYTE desenvolvido aqui, e no próximo capítulo iremos modificar nosso arquivo para eliminar esses problemas de tecla de função.

Entrada Hexa

Vamos começar reescrevendo LE_BYTE. No capítulo anterior, LE_BYTE leria qualquer caractere comum ou tecla de função, e retornaria um byte para Controle. Controle, então, chamava o Editor se LE_BYTE tivesse lido um caractere comum, e EDITA_BYTE modificaria o byte apontado pelo cursor fantasma. Caso contrário, Controle procurava os códigos na tabela TAB_CONTROLE, verificando se o byte estava lá; se estivesse, Controle chamaria a rotina indicada na tabela.

Mas, como dissemos no capítulo anterior, a antiga versão de LE_BYTE torna muito fácil a alteração de um byte por acidente. Se você inadvertidamente esbarrar em uma tecla do teclado (além das teclas especiais), EDITA_BYTE imediatamente entrará em ação, modificando o byte na posição do cursor. Todos nós podemos nos distrair, e uma distração dessas pode ter resultados desastrosos no setor do disco.

Vamos alterar LE_BYTE de modo que, daqui por diante, ele não envie o caractere que digitamos até que seja pressionada a tecla Enter. Essa característica será incluída com o auxílio da função 0Ah da interrupção INT 21h do DOS, que lê um string de caracteres. O DOS retorna esse string somente quando pressionarmos a tecla Enter, para que possamos ter mudanças antidistração. Mas com isso nós perdemos as teclas de função, por motivos que veremos mais adiante.

Para ver exatamente como nossas mudanças afetam LE_BYTE, precisamos escrever um programa de teste para testar LE_BYTE em isolado. Dessa forma, se acontecer algo estranho, saberemos que LE_BYTE é o culpado, e não uma outra parte qualquer de Dskpatch. Nossa tarefa será escrever uma rotina de teste que será mais simples se usarmos algumas poucas rotinas de Es_tec, Es_video e Cursor para imprimir a informação sobre o progresso de LE_BYTE. Usaremos rotinas como IMP_HEXA e IMP_DECIMAL para imprimir o código do caractere retornado e o número de caracteres lido. Os detalhes estão aqui, em TESTE.ASM:

Listagem 23-1. O Programa de Teste TESTE.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        ORG     100h

        EXTRN   IMP_HEXA:NEAR, IMP_DECIMAL:NEAR
        EXTRN   IMP_STRING:NEAR, ENVIA_CRLF:NEAR
        EXTRN   LE_BYTE:NEAR

TESTE   PROC    NEAR
        LEA     DX,PRONTO_ENTRADA
```

```
        CALL    IMP_STRING
        CALL    LE_BYTE
        CALL    ENVIA_CRLF
        LEA     DX,PRONTO_CARACTERE
        CALL    IMP_STRING
        MOV     DL,AL
        CALL    IMP_HEXA
        CALL    ENVIA_CRLF
        LEA     DX,PRONTO_CARACT
        CALL    IMP_STRING
        MOV     DL,AH
        XOR     DH,DH
        CALL    IMP_DECIMAL
        CALL    ENVIA_CRLF
        INT     20h
TESTE   ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
PRONTO_ENTRADA          DB      'Entre caracteres: ',0
PRONTO_CARACTERE        DB      'Codigo do caractere: ',0
PRONTO_CARACT_LIDOS     DB      'Numero de caracteres lidos: ',0
; variaveis nao usadas
        PUBLIC  NR_LINHA_CAB, NR_DRIVE, CAB_1, CAB_2
        PUBLIC  NR_LINHA_PRONTO, NR_SETOR
NR_LINHA_CAB            DB      0
NR_DRIVE                DB      0
CAB_1                   DB      0
CAB_2                   DB      0
NR_LINHA_PRONTO         DB      0
NR_SETOR                DB      0
DATA_SEG        ENDS

        END     TESTE
```

Tente linkeditar este programa com nossas versões atuais de Es_tec, Es_video e Cursor (coloque Teste em primeiro lugar na lista de LINK). Se você pressionar qualquer tecla de função especial, Teste dirá que você leu 255 caracteres. Por quê? Colocamos o −1 de AH em DL e zeramos o byte mais alto de DX, deixando DX com 255 (FFh), e não com −1 (FFFFh).

Não iremos nos distrair dessa forma quando usarmos a versão de LE_BYTE em Dskpatch. Esse é um programa de teste, e enquanto soubermos o que esperar, podemos testar LE_BYTE e todas as suas condições de limite. Antes de continuarmos a escrever LE_BYTE, entretanto, precisamos explicar uma característica de TESTE.ASM que você pode ter notado: suas definições de variável.

A parte principal das instruções em TESTE.ASM são para a formatação — tornar a tela mais agradável visualmente. As definições de variável no final de Teste são incluídas só para satisfazer o linkeditor. Quando linkeditamos Teste com Es_tec, Es_video e Cursor, o linkeditor procura uma série de variáveis usadas pelos últimos três arquivos. Nós definimos as variáveis em Dskpatch, mas como não estamos linkeditando em Dskpatch, precisamos redefinir essas variáveis em TESTE.ASM. Não vamos usar as variáveis, pois não estamos chamando rotinas (em Es_video e Cursor) que precisem delas. Mas precisamos dessas variáveis de qualquer forma, para que o linkeditor não "arranque os cabelos".

Vamos prosseguir em direção a LE_BYTE aceitando um string de caracteres. Isso não somente evitará distrações quando estivermos usando o Dskpatch, como também nos permitirá usar a tecla de retrocesso (*Backspace*) para apagar caracteres que já digitamos — outra característica importante, pois é muito fácil cometermos erros. LE_BYTE usará a rotina LE_STRING para ler um string de caracteres.

LE_STRING é muito simples, quase trivial, mas a colocamos em uma rotina separada para que pudesse ser reescrita no próximo capítulo, de modo que leia teclas de função especial sem a necessidade da tecla Enter. Para economizar tempo, também incluiremos três rotinas utilizadas por LE_BYTE: STRING_MAIUSC, CONVERTE_DIG_HEXA e HEXA_PARA_BYTE.

STRING_MAIUSC e HEXA_PARA_BYTE trabalham ambos com strings. STRING_MAIUSC converte todas as letras minúsculas em um string de maiúsculas. Isso significa que você pode digitar f3 ou F3 para o número hexa F3h. Permitindo o uso de ambas as formas em nosso programa, Dskpatch torna-se um sistema amigo do usuário.

HEXA_PARA_BYTE apanha um string lido pelo DOS, após chamarmos STRING_MAIUSC, e converte o string hexa de dois dígitos para um número de único byte. HEXA_PARA_BYTE utiliza CONVERTE_DIG_HEXA para fazer a conversão de cada dígito hexa em um número de quatro bits.

Como podemos ter certeza de que o DOS não lerá mais do que dois dígitos hexa? A função 0Ah da interrupção INT 21h do DOS lê um string inteiro de caracteres para a área de memória definida da seguinte forma:

```
LIMITE_NUM_CARACT   DB    0
NUM_CARACT_LIDOS    DB    0
CHARS               DB    80 DUP (0)
```

O primeiro byte evita que entremos com muitos caracteres. LIMITE_NUM_CARACT diz ao DOS quantos caracteres, no máximo, podem ser lidos. Se colocarmos três, o DOS lerá somente dois caracteres, e mais o caractere de retorno-de-carro (o DOS sempre conta o retorno-de-carro). Quaisquer caracteres digitados após esse limite serão desprezados, sendo que o DOS emite um bipe para avisar que passamos do limite. Quando pressionarmos a tecla Enter, o DOS colocará o número de caracteres realmente lidos no segundo byte, NUM_CARACT_LIDOS, sem incluir o retorno-de-carro.

Ambas as rotinas STRING_MAIUSC e LE_BYTE usam NUM_CARACT_LIDOS. Por exemplo, LE_BYTE verifica NUM_CARACT_LIDOS para descobrir se você digitou somente um caractere ou um número hexa de dois dígitos. Se NUM_CARACT_LIDOS possuir o valor 1, LE_BYTE retorna um único caractere no registrador AL. Se NUM_CARACT_LIDOS possuir o valor 2, LE_BYTE usa HEXA_PARA_BYTE a fim de converter o string hexa de dois dígitos em um único byte.

Sem mais demora, veja o novo arquivo ES_TEC.ASM, com todas as quatro novas rotinas:

Listagem 23-2. A Nova Versão de ES_TEC.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  STRING_MAIUSC

;-----------------------------------------------------------------------;
; Esta rotina converte o string, usando o formato do DOS para strings,  ;
; em letras maiúsculas.                                                 ;
;                                                                       ;
;       DS:DX     Endereço do buffer do string                          ;
;-----------------------------------------------------------------------;
STRING_MAIUSC   PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX                      ;Contador de caracteres
```

```
        MOV     CL,[BX]                 ;no segundo byte do buffer
        XOR     CH,CH                   ;Zera byte alto do contador
LOOP_MAIUSC:
        INC     BX                      ;Aponta prox. caractere
        MOV     AL,[BX]
        CMP     AL,'a'                  ;Ve se e letra minuscula
        JB      NAO_MINUSC              ;Nao
        CMP     AL,'z'
        JA      NAO_MINUSC
        ADD     AL,'A'-'a'              ;Converte para maiuscula
        MOV     [BX],AL
NAO_MINUSC:
        LOOP    LOOP_MAIUSC
        POP     CX
        POP     BX
        POP     AX
        RET
STRING_MAIUSC           ENDP

;-------------------------------------------------------------------------------
; Esta rotina converte um caractere de ASCII (hexa) para um nibble
; (4 bits)
;
;           AL      Caractere a converter
; Retorna:  AL      Nibble
;           CF      Setado se houver erro
;-------------------------------------------------------------------------------
CONVERTE_DIG_HEXA       PROC    NEAR
        CMP     AL,'0'                  ;Digito legal?
        JB      ERRO_DIG                ;Nao
        CMP     AL,'9'                  ;Nao esta certo ainda
        JA      TENTA_HEXA              ;Poderia ser digito hexa
        SUB     AL,'0'                  ;Se digito decimal, converte para nibble
        CLC                             ;Limpa o carry, nenhum erro
        RET
TENTA_HEXA:
        CMP     AL,'A'                  ;Nao tem certeza
        JB      ERRO_DIG                ;Nao hexa
        CMP     AL,'F'                  ;Nao tem certeza
        JA      ERRO_DIG                ;Nao hexa
        SUB     AL,'A'-10               ;Hexa, converte para nibble
        CLC                             ;Limpa o carry
        RET
ERRO_DIG:
        STC                             ;Seta o carry, erro
        RET
CONVERTE_DIG_HEXA       ENDP

        PUBLIC  HEXA_PARA_BYTE

;-------------------------------------------------------------------------------
; Esta rotina converte os dois caracteres em DS:DX de hexa para
; um byte.
;
;       DS:DX   Endereco dos dois caracteres do numero hexa
; Retorna:
;       AL      Byte
;       CF      Setado se houver erro
;
; Usa:          CONVERTE_DIG_HEXA
;-------------------------------------------------------------------------------
HEXA_PARA_BYTEPROC      NEAR
        PUSH    BX
        PUSH    CX
        MOV     BX,DX                   ;Coloca endereco em BX para endereco indireto
        MOV     AL,[BX]                 ;Apanha primeiro digito
        CALL    CONVERTE_DIG_HEXA
        JC      ERRO_HEXA               ;Digito hexa errado se flag setado
```

```
        MOV     CX,4                    ;Agora multiplica por 16
        SHL     AL,CL
        MOV     AH,AL                   ;Retem uma copia
        INC     BX                      ;Apanha segundo digito
        MOV     AL,[BX]
        CALL    CONVERTE_DIG_HEXA
        JC      ERRO_HEXA               ;Digito hexa errado se carry setado
        OR      AL,AH                   ;Combina dois nibbles
        CLC                             ;Limpa carry se nao houver erro
FIM_HEXA:
        POP     CX
        POP     BX
        RET
ERRO_HEXA:
        STC                             ;Seta carry, indicando erro
        JMP     FIM_HEXA
HEXA_PARA_BYTE  ENDP
;-------------------------------------------------------------------------
; Esta e uma versao simples de LE_STRING                                  ;
;                                                                         ;
;       DS:DX   Endereco da area de string                                ;
;-------------------------------------------------------------------------
LE_STRING       PROC    NEAR
        PUSH    AX
        MOV     AH,0Ah                  ;Chama entrada bufferizada do teclado
        INT     21h                     ;Chama funcao do DOS para entrada bufferizada
        POP     AX
        RET
LE_STRING       ENDP

        PUBLIC  LE_BYTE

;-------------------------------------------------------------------------
; Esta rotina le ou um caractere ASCII ou um numero hexa                  ;
; de dois digitos. Esta e uma versao de teste de LE_BYTE.                 ;
;                                                                         ;
; Retorna byte em   AL    Codigo do caractere (a nao ser que AH = 0)      ;
;                   AH    1 se leu caractere ASCII                        ;
;                         0 se nao leu caracteres                         ;
;                         -1 se leu uma tecla especial                    ;
;                                                                         ;
; Usa:      HEXA_PARA_BYTE, STRING_MAIUSC, LE_STRING                      ;
; Le:       ENTRA_TECLADO, etc.                                           ;
; Escreve:  ENTRA_TECLADO, etc.                                           ;
;-------------------------------------------------------------------------

LE_BYTE         PROC    NEAR
        PUSH    DX
        MOV     LIMITE_NUM_CARACT,3     ;Permite somente dois caracteres (mais Enter)
        LEA     DX,ENTRA_TECLADO
        CALL    LE_STRING
        CMP     NUM_CARACT_LIDOS        ;Ve quantos caracteres
        JE      ENTROU_ASCII            ;So um, trata como caractere ASCII
        JB      SEM_CARACT              ;Somente tecla Enter
        CALL    STRING_MAIUSC           ;Nao, converte string para maiusculas
        LEA     DX,STRING               ;Endereco do string a converter
        CALL    HEXA_PARA_BYTE          ;Converte string de hexa para byte
        JC      SEM_CARACT              ;Erro, retorna nenhum caractere lido
        MOV     AH,1                    ;Indica que leu um caractere
FIM_LEITURA:
        POP     DX
        RET
SEM_CARACT:
        XOR     AH,AH                   ;Indica 'nenhum caractere lido'
        JMP     FIM_LEITURA
ENTROU_ASCII:
        MOV     AL,STRING               ;Carrega caractere lido
```

```
        MOV     AH,1                    ;Indica que leu um caractere
        JMP     FIM_LEITURA
LE_BYTE         ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
ENTRA_TECLADO   LABEL   BYTE
LIMITE_NUM_CARACT   DB      0           ;Tamanho do buffer de entrada
        NUM_CARACT_LIDOS    DB      0   ;Nº de caracteres lidos
STRING          DB      80 DUP (0)      ;Um buffer para entrada pelo teclado
DATA_SEG        ENDS

        END
```

Monte novamente Es_tec e linkedite os quatro arquivos Teste, Es_tec, Es_video e Cursor, para executar essa versão de LE_BYTE.

Nesse ponto, há dois problemas com LE_BYTE. Lembra-se das teclas de função especial? Não podemos lê-las com a função 0Ah do DOS. Simplesmente não funciona. Tente pressionar uma tecla de função quando executar o programa Teste. O DOS não retorna dois bytes, com o primeiro zerado, como você poderia esperar.

Não temos um meio de ler os códigos estendidos com a entrada bufferizada de teclado pelo DOS, usando a função 0Ah. Usamos essa função para que pudéssemos usar a tecla de retrocesso para deletar caracteres antes que pressionássemos a tecla Enter. Mas agora, como não podemos ler as teclas de função especial, temos que escrever nossa própria rotina LE_STRING. Temos que trocar a função 0Ah para que uma tecla de função possa ser pressionada sem a necessidade do Enter.

O outro problema com a função 0Ah do DOS tem a ver com o caractere de mudança-de-linha. Pressione a tecla Control-Enter (mudança-de-linha) depois de digitar um caractere, e aí então tente a tecla de retrocesso. Você se encontrará na linha seguinte, sem um modo de retornar à anterior. Nossa nova versão de Es_tec no capítulo seguinte cuidará do caractere de mudança-de-linha (Control-Enter) como um caractere comum; depois disso, se pressionarmos a mudança-de-linha o cursor não moverá para a linha seguinte.

Mas antes de resolvermos os problemas com LE_BYTE e LE_STRING, vamos escrever uma rotina para ler um número decimal sem sinal. Não vamos usar a rotina neste livro, mas ela pode ser importante se você quiser acrescentar certas facilidades ao programa; por exemplo, pedindo ao Dskpatch para apresentar o setor número 567.

Entrada Decimal

Lembre-se de que o maior número decimal não sinalizado que podemos colocar em uma única palavra é 65535. Quando usarmos LE_STRING para ler um string de dígitos decimais, diremos ao DOS para ler não mais do que seis caracteres (cinco dígitos e um retorno-de-carro no final). Logicamente, isso significa que LE_DECIMAL poderá ler números de 65536 até 99999, mesmo que esses números não possam ser representados em somente uma palavra. Teremos que tomar conta desses números e retornar um código de erro se LE_DECIMAL tentar ler um número maior do que 65535, ou se tentar ler um caractere que não está entre zero e nove.

Para converter nosso string de até cinco dígitos em uma palavra binária, usamos a multiplicação da mesma forma que no Capítulo 1: pegue o primeiro dígito (o da esquerda), multiplique-o por *dez*

à quarta potência, acrescente o segundo dígito multiplicado por dez à terceira potência, e assim por diante. Usando esse método, poderíamos, por exemplo, escrever 49856 como:

$4*10^4 + 9*10^3 + 8*10^2 + 5*10^1 + 6*10^0$

ou, conforme faremos nos cálculos da rotina:

10*(10*(10*(10*4+9) +8) +5) +6

Na verdade, devemos tomar cuidado com os erros nestas contas de multiplicação e retornar um aviso, com o flag de carry setado, quando ocorrer algum erro. Como podemos saber que estamos lendo um número maior do que 65535? Com números maiores, a última conta de multiplicação (o último MUL) irá estourar o registrador DX. O flag de carry é setado quando DX não é zero após a palavra MUL, de modo que podemos usar um JC (*Jump if Carry set — Salta se Carry setado*) para lidar com uma condição de erro. Eis a rotina IMP_DECIMAL, que também verifica se houve um erro no dígito (um dígito que não está entre 0 e 9). Coloque essa rotina no arquivo ES_TEC.ASM:

Listagem 23-3. Acrescente Esta Rotina em ES_TEC.ASM

```
        PUBLIC  LE_DECIMAL

;-------------------------------------------------------------------------
; Esta rotina apanha o buffer de saida de LE_STRING e converte
; o string de digitos decimais para uma palavra.
;
;       AX      Palavra convertida de decimal
;       CF      Setado se houver erro
;
; Usa:          LE_STRING
; Le:           ENTRA_TECLADO, etc.
; Escreve:      ENTRA_TECLADO, etc.
;-------------------------------------------------------------------------
LE_DECIMAL      PROC    NEAR
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     LIMITE_NUM_CARACT,6     ;Numero maximo 5 digitos (65535)
        LEA     DX,ENTRA_TECLADO
        CALL    LE_STRING
        MOV     CL,NUM_CARACT_LIDOS     ;Apanha numero de caracteres lidos
        XOR     CH,CH                   ;Zera byte alto do contador
        CMP     CL,0                    ;Retorna erro se nenhum caractere lido
        JLE     ERRO_DIG_DECIMAL
        XOR     AX,AX                   ;Comeca com numero zero
        XOR     BX,BX                   ;e no inicio do string
CONVERTE_DIGITO:
        MOV     DX,10                   ;Multiplica numero por 10
        MUL     DX                      ;Multiplica AX por 10
        JC      ERRO_DIG_DECIMAL        ;CF setado se MUL estourou uma palavra
        MOV     DL,STRING[BX]           ;Apanha proximo digito
        SUB     DL,'0'                  ;E converte para um nibble (4 bits)
        JS      ERRO_DIG_DECIMAL        ;Digito errado se < 0
        CMP     DL,9                    ;E um digito errado?
        JA      ERRO_DIG_DECIMAL        ;Sim
        ADD     AX,DX                   ;Nao, entao soma ao numero
        INC     BX                      ;Aponta proximo caractere
        LOOP    CONVERTE_DIGITO         ;Apanha proximo digito
FIM_DECIMAL:
        POP     DX
        POP     CX
        POP     BX
        RET
ERRO_DIG_DECIMAL:
        STC                             ;Seta carry para avisar erro
        JMP     FIM_DECIMAL
LE_DECIMAL      ENDP
```

Para termos certeza de que essa rotina funciona corretamente, precisamos testá-la com todas as condições. Eis um programa simples de teste para LE_DECIMAL, que usa o mesmo método de LE_BYTE:

Listagem 23-4. Alterações em TESTE.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:GROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        ORG     100h

        EXTRN   IMP_HEXA:NEAR, IMP_DECIMAL:NEAR
        EXTRN   IMP_STRING:NEAR, ENVIA_CRLF:NEAR
        EXTRN   LE_DECIMAL:NEAR

TESTE   PROC    NEAR
        LEA     DX,PRONTO_ENTRADA
        CALL    IMP_STRING
        CALL    LE_DECIMAL
        JC      ERRO
        CALL    ENVIA_CRLF
        LEA     DX,PRONTO_NUM_LIDOS
        CALL    IMP_STRING
        MOV     DX,AX
        CALL    IMP_DECIMAL
ERRO:   CALL    ENVIA_CRLF
        INT     20h
TESTE   ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
PRONTO_ENTRADA          DB
                        DB

        PUBLIC  NR_LINHA_CAB, NR_DRIVE, CAB_1, CAB_2
        PUBLIC  NR_LINHA_PRONTO, NR_SETOR
NR_LINHA_CAB            DB
                        DB
                        DB
                        DB
                        DB
                        DB
                ENDS

        END     TESTE
```

Novamente, precisamos linkeditar quatro arquivos: Teste (o primeiro da lista), Es_tec, Es_video e Cursor. Tente as condições de limite usando dígitos válidos e não-válidos (como o A, que não é válido como dígito decimal), e com números como 0, 65535 e 65536.

Resumo

Voltaremos às duas rotinas simples de teste mais adiante, quando falarmos sobre alguns modos de você escrever seus próprios programas. Depois veremos como usar uma versão ligeiramente mais avançada de TESTE.ASM para escrever um programa que fará a conversão de números hexa e decimal.

Por enquanto, estamos prontos para continuar no capítulo seguinte, onde escreveremos versões melhoradas de LE_BYTE e LE_STRING.

24

ENTRADA DE TECLADO MELHORADA

Um Novo LE_STRING

Já tínhamos dito que iríamos apresentar o desenvolvimento de Dskpatch conforme o escrevemos inicialmente — incluindo erros e rotinas desajeitadas, e algumas delas você já viu. Neste capítulo, vamos escrever uma nova versão de LE_BYTE, mas ficará com um erro malicioso em Dskpatch. (Em inglês, esse tipo de erro é chamado de *"bug"* — inseto —, conforme uma lenda existente na área de processamento de dados.) No próximo capítulo, vamos providenciar um frasco de inseticida para eliminar esse pequeno "bug", mas veja se pode descobri-lo por si mesmo. (Dica: Verifique cuidadosamente os limites de condição em LE_BYTE quando ele for unido a Dskpatch.)

Um Novo LE_STRING

Nossa filosofia de projeto modular pede rotinas pequenas, e por isso nenhuma rotina sozinha é difícil de ser entendida. A nova versão de LE_STRING será um exemplo de rotina desajeitada: muito longa. Ela deve ser escrita com mais rotinas, mas deixamos isso para você. Este livro está quase chegando a um final, mas ainda deixamos algumas rotinas que devem ser feitas antes que Dskpatch se torne um programa útil. Nesse momento, podemos editar somente a primeira metade de qualquer setor, e ainda não podemos gravar esse setor de volta no disco.

Neste capítulo, daremos a LE_STRING uma nova rotina, BACK_SPACE, para emular a função de retrocesso encontrada na função 0Ah do DOS. Quando pressionarmos a tecla Backspace, BACK_SPACE apagará o último caractere digitado, tanto na tela quanto no string armazenado na memória.

Na tela, BACK_SPACE apagará o caractere movendo o cursor uma posição à esquerda, imprimindo um espaço sobre ele, e depois movendo o cursor uma posição à esquerda novamente. Essa seqüência executará a mesma função de retrocesso executada pelo DOS.

No buffer, BACK_SPACE apagará um caractere alterando o ponteiro do buffer, DS:SI+BX, de modo que aponte para o próximo byte mais alto na memória. Em outras palavras, BACK_SPACE simplesmente decrementará BX (BX = BX − 1). O caractere ainda estará no buffer, mas o programa não o verá. Por que não? A rotina LE_STRING nos diz quantos caracteres estão sendo lidos. Se tentarmos ler mais do que esse número no buffer, veremos os caracteres apagados. Caso contrário, não veremos.

Temos que ter o cuidado de não apagar qualquer caractere quando o buffer estiver vazio. Lembre-se que nossa área de dados era algo assim:

```
LIMITE_NUM_CARACT   DB    0
NUM_CARACT_LIDOS    DB    0
STRING              DB    80 DUP (0)
```

O buffer do string começa no segundo byte dessa área de dados, ou no *deslocamento* 2 a partir do início (os endereços começam a ser contados de 0). Assim, BACK_SPACE não apagará um caractere se BX for 2, o início do buffer de string, pois o buffer está vazio quando BX é igual a 2.

Veja, a seguir, BACK_SPACE; coloque-o em ES_TEC.ASM:

Listagem 24-1. Acrescente Esta Rotina em ES_TEC.ASM

```
        PUBLIC  BACK_SPACE
        EXTRN   IMP_CAR:NEAR
```

```
;-----------------------------------------------------------------------
; Esta rotina deleta um caractere de cada vez, do buffer e da tela
; quando o buffer nao esta vazio. BACK_SPACE simplesmente
; retorna quando o buffer esta vazio.
;
;     DS:SI+BX        Caractere mais recente do buffer
;
; Usa:            IMP_CAR
;-----------------------------------------------------------------------

BACK_SPACE      PROC    NEAR    ;Deleta um caractere
        PUSH    AX
        PUSH    DX
        CMP     BX,2            ;Buffer vazio?
        JE      FIM_BS          ;Sim, le o prox. caractere
        DEC     BX              ;Remove um caractere do buffer
        MOV     AH,2            ;Remove caractere da tela
        MOV     DL,BS
        INT     21h
        MOV     DL,20h          ;Imprime espaco aqui
        CALL    IMP_CAR
        MOV     DL,BS           ;Retrocede
        INT     21h
FIM_BS: POP     DX
        POP     AX
        RET
BACK_SPACE      ENDP
```

Vamos continuar com a nova versão de LE_STRING. Ele está bem grande; a listagem é somente para uma rotina. Disparado é a maior rotina que já escrevemos. Ela é grande e complicada por ter muitas condições possíveis.

Por que LE_STRING faz tantas coisas? Acrescentamos mais algumas características. Se você pressionar a tecla Escape, LE_STRING apagará o buffer e removerá todos os caracteres da tela. O DOS também apaga todos os caracteres do buffer quando você pressiona Escape, mas não apaga caracteres na tela. No lugar disso, ele simplesmente coloca uma barra invertida (\ ) no final da linha e move para a linha seguinte. Nossa versão de LE_STRING será mais versátil do que a função do DOS.

LE_STRING usa três teclas especiais: o Backspace, Escape e a tecla de Enter. Poderíamos escrever os códigos ASCII dessas teclas em LE_STRING, quando precisássemos delas, mas, ao invés disso, vamos colocar algumas definições no início de ES_TEC.ASM para tornar LE_STRING mais legível. Eis as definições:

Listagem 24-2. Acréscimos a ES_TEC.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

BS      EQU     8
                                ;Caractere de backspace
CR      EQU     13
                                ;Caractere retorno-de-carro
ESC     EQU     27
                                ;Caractere Escape

CODE_SEG        SEGMENT PUBLIC
```

Veja em seguida LE_STRING. Embora seja um tanto longo, você pode ver que não é tão complicado — somente longo. Troque a versão antiga de LE_STRING em ES_TEC.ASM por essa nova versão:

Listagem 24-3. O Novo LE_STRING em ES_TEC.ASM

```
        PUBLIC  LE_STRING
        EXTRN   IMP_CAR:NEAR

;-----------------------------------------------------------------------
; Esta rotina e semelhante a funcao 0Ah do DOS. Mas essa
; funcao retorna um caractere especial se uma tecla de
; funcao for pressionada -- sem retornar essas teclas.
; E ESC apagara a entrada e comecara de novo.
;
;       DS:DX   Endereco do buffer do teclado. O primeiro byte deve
;               conter um maximo de caracteres a ler (mais
;               o CR). E o segundo byte sera usado por esta
;               rotina para retornar o numero de caracteres
;               realmente lidos.
;
;                       0       Nenhum caractere lido
;                       -1      Um caractere lido de
;                       outra forma, numero realmente lido (sem a
;                               tecla Enter)
;
;  Usa:         BACK_SPACE, IMP_CAR
;-----------------------------------------------------------------------

LE_STRING       PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    SI
        MOV     SI,DX                   ;Usa SI para reg. de indice e
INICIO:
        MOV     BX,2                    ;BX para deslocamento no buffer
        MOV     AH,7                    ;Chama para entrada sem checar
        INT     21h                     ;CTRL_BREAK e sem eco
        OR      AL,AL                   ;E um caractere estendido?
        JZ      ESTENDIDO               ;Sim, le o caractere estendido
NAO_ESTENDIDO:                          ;Caractere estendido e erro se buffer nao vazio
        CMP     AL,CR                   ;E um retorno de carro (CR)?
        JE      FIM_ENTRADA             ;Sim, fim da entrada
        CMP     AL,BS                   ;E um retrocesso (BS)?
        JNE     NAO_BS                  ;Nao
        CALL    BACK_SPACE              ;Sim, deleta caractere
        CMP     BL,2                    ;Buffer esta vazio?
        JE      INICIO                  ;Sim, pode ler caracteres estendidos de novo
        JMP     SHORT LE_PROX_CAR       ;Nao, continua lendo normalmente
NAO_BS: CMP     AL,ESC                  ;E um ESC -- apagar buffer?
        JE      APAGA_BUFFER            ;Sim, apaga o buffer
        CMP     BL,[SI]                 ;Verifica se buffer esta cheio
        JA      BUFFER_CHEIO            ;Buffer cheio
        MOV     [SI+BX],AL              ;Senao, salva caractere no buffer
        INC     BX                      ;Aponta prox. byte livre no buffer
        PUSH    DX
        MOV     DL,AL                   ;Ecoa caractere na tela
        CALL    IMP_CAR
        POP     DX
LE_PROX_CAR:
        MOV     AH,7
        INT     21h
        OR      AL,AL                   ;Um caractere estendido nao e valido
                                        ;   quando o buffer nao esta vazio
        JNE     NAO_ESTENDIDO           ;Caractere valido
        MOV     AH,7
        INT     21h                     ;Elimina caractere estendido

;-----------------------------------------------------------------------
; Indica uma condicao de erro por meio de
; um beep no console: chr$(7).
;-----------------------------------------------------------------------

INDICA_ERRO:
        PUSH    DX
```

```
        MOV     DL,7                    ;Avisa com beep (chr$(7))
        MOV     AH,2
        INT     21h
        POP     DX
        JMP     SHORT LE_PROX_CAR       ;Agora le o proximo caractere

;-------------------------------------------------------------------------
; Esvazia o buffer de string e apaga todos
; os caracteres apresentados na tela.
;-------------------------------------------------------------------------
APAGA_BUFFER:
        PUSH    CX
        MOV     CL,[SI]                 ;Retrocede numero maximo de
        XOR     CH,CH
LOOP_APAGA:                             ;  caracteres no buffer. BACK_SPACE
        CALL    BACK_SPACE              ;evitara que o cursor se
        LOOP    LOOP_APAGA              ;  mova muito para tras
        MOV     CX
        JMP     INICIO                  ;Agora pode ler caracteres ASCII estendidos,
                                        ;pois o buffer esta vazio

;-------------------------------------------------------------------------
; O buffer estava cheio, e nao podemos ler
; outro caractere. Envia um beep para alertar
; o usuario da condicao do buffer.
;-------------------------------------------------------------------------
BUFFER_CHEIO:
        JMP     SHORT INDICA_ERRO       ;Se buffer cheio, sinaliza

;-------------------------------------------------------------------------
; Le o codigo estendido e o coloca no buffer
; como unico caractere, e depois retorna
; -1 como numero de caracteres lidos.
;-------------------------------------------------------------------------
ESTENDIDO:                              ;Le um codigo ASCII estendido
        MOV     AH,7
        INT     21h
        MOV     [SI+2],AL               ;Coloca esse caractere no buffer
        MOV     BL,0FFh                 ;Num. caract. lidos = -1 para especiais
        JMP     SHORT FIM_STRING

;-------------------------------------------------------------------------
; Salva a contagem do numero de caracteres
; lidos e retorna.
;-------------------------------------------------------------------------
FIM_ENTRADA:                            ;Termina entrada
        SUB     BL,2                    ;Contador de caracteres lidos
FIM_STRING:
        MOV     [SI+1],BL               ;Retorna numero de caracteres
        POP     SI
        POP     BX
        POP     AX
        RET
LE_STRING       ENDP
```

Analisando a rotina, podemos ver que ela primeiro verifica se pressionamos uma tecla de função especial. Ela nos permite fazer isso só quando o string estiver vazio. Por exemplo, se pressionarmos F1 após termos pressionado a tecla *a*, LE_STRING ignorará a tecla F1 e gerará um bipe dizendo que pressionamos uma tecla especial na hora errada. Podemos, entretanto, pressionar Escape e depois a tecla F1, pois a tecla Esc limpa o buffer do string.

Se LE_STRING encontrar um retorno-de-carro, ele colocará o número de caracteres lidos no segundo byte da área da memória que lida com a entrada de strings, e retornará. Nossa versão de LE_BYTE verifica nesse byte o número de caracteres que foram realmente lidos.

Em seguida, a rotina verifica se digitamos um caractere de retrocesso. Caso afirmativo, ela chama BACK_SPACE para apagar o caractere. Se o buffer do string ficar vazio (BX se tornar igual a 2, que é o início do buffer do string), então LE_STRING volta ao início, onde poderá ler uma tecla especial. Caso contrário, a rotina simplesmente lerá o próximo caractere.

Finalmente, LE_STRING verifica o caractere Esc. BACK_SPACE só apaga os caracteres quando houver caracteres no buffer, de modo que podemos limpar o buffer do string chamando BACK_SPACE por LIMITE_NUM_CARACT vezes, pois LE_STRING nunca poderá ler mais do que LIMITE_NUM_CARACT caracteres. Qualquer outro caractere será armazenado no buffer e ecoado na tela com IMP_CAR. A não ser, é claro, que o buffer esteja cheio.

No capítulo anterior, alteramos LE_BYTE de uma forma tal que ele não poderia ler teclas de função especial. Precisamos somente acrescentar algumas linhas para permitir que LE_BYTE trabalhe com nossa versão de LE_STRING, que pode ler teclas de função especial. Eis as mudanças a fazer em LE_BYTE, de ES_TEC.ASM:

Listagem 24-4. Alterações em LE_BYTE, de ES_TEC.ASM

```
        PUBLIC  LE_BYTE
;-----------------------------------------------------------------------------
; Esta rotina le um unico caractere ASCII ou um numero hexa de 2 digitos.
;
; Retorna byte em       AL      Codigo do caractere (a menos que AH = 0)
;                       AH      1 se leu caract. ASCII ou numero hexa
;                               0 se nenhum caractere
;                               -1 se leu tecla especial
;
; Usa:          HEXA_PARA_BYTE, STRING_MAIUSC, LE_STRING
; Le:           ENTRA_TECLADO, etc.
; Escreve:      ENTRA_TECLADO, etc.
;-----------------------------------------------------------------------------
LE_BYTE         PROC    NEAR
        PUSH    DX
        MOV     LIMITE_NUM_CARACT,3             ;Permite somente 2 caracteres (mais Enter)
        LEA     DX,ENTRA_TECLADO
        CALL    LE_STRING
        CMP     NUM_CARACT_LIDOS,1              ;Ve quantos caracteres
        JE      ENTRADA_ASCII                   ;So um, tratar como caractere ASCII
        JB      SEM_CARACTERES                  ;So ENTER pressionado
        CMP     BYTE PTR NUM_CARACT_LIDOS,0FFh  ;Tecla especial?
        JE      TECLA_ESPECIAL                  ;Sim
        CALL    STRING_MAIUSC                   ;Nao, converte para maiusculas
        LEA     DX,STRING                       ;Endereco do string a converter
        CALL    HEXA_PARA_BYTE                  ;Converte string de hexa para byte
        JC      SEM_CARACTERES                  ;Erro, retorna nenhum caractere lido
        MOV     AH,1                            ;Indica que leu um caractere
FIM_LEITURA:
        POP     DX
        RET
SEM_CARACTERES:
        XOR     AH,AH                           ;Indica 'nenhum caractere lido'
        JMP     FIM_LEITURA
ENTRADA_ASCII:
        MOV     AL,STRING                       ;Carrega caractere lido
        MOV     AH,1                            ;Indica que leu 1 caractere
        JMP     FIM_LEITURA
TECLA_ESPECIAL:
        MOV     AL,STRING[0]                    ;Retorna codigo de varredura
        MOV     AH,0FFh                         ;Indica tecla especial com -1
        JMP     FIM_LEITURA
LE_BYTE         ENDP
```

Dskpatch, com as novas versões de LE_BYTE e LE_STRING, deve agora estar muito melhor de se usar. Mas há um "bug" aqui, como dissemos. Para tentar descobri-lo, execute Dskpatch e teste todas as condições de limite para LE_BYTE e HEXA_PARA_BYTE.

25

À PROCURA DE ERROS

Consertando CONTROLA

Resumo

Se você tentar usar a nova versão de Dskpatch com *ag*, que não é um número hexa, notará que Dskpatch não faz nada quando você pressiona a tecla Enter. Como o string *ag* não é um número hexa, o programa deveria, pelo menos, apagá-lo da tela.

Este é um tipo de erro que podemos descobrir só testando exaustivamente as condições de limite de um programa. Não apenas as partes, mas o programa inteiro. O erro aqui não é culpa de LE_BYTE, mesmo que tenha aparecido quando escrevemos essa rotina. Ao invés disso, o problema está no modo com que escrevemos CONTROLA e EDITA_BYTE.

EDITA_BYTE foi planejado de modo que chamasse IMP_PRONTO e reimprimisse a linha de pronto, apagando o restante da linha. Isso removerá qualquer caractere que digitamos. Mas se digitarmos um string como *ag*, LE_BYTE informa que leu um string de tamanho zero, e CONTROLA não chama EDITA_BYTE. Qual é a solução?

Consertando CONTROLA

Há, na verdade, dois modos de se resolver esse problema. A melhor solução seria reescrever Dskpatch de forma mais modular, e reprojetar CONTROLA. Não faremos isso. Lembre-se: os programas nunca estão completos, mas temos que parar em algum ponto. Ao invés disso, acrescentaremos um reparo em CONTROLA, para que reimprima o pedido de comando (pronto) sempre que LE_BYTE ler um string de tamanho zero.

Eis as modificações em CONTROLA (no arquivo CONTROLE.ASM) para "dar um spray" nesse "bug":

Listagem 25-1. Alterações em CONTROLA, de CONTROLE.ASM

```
        PUBLIC  CONTROLA
        EXTRN   LE_BYTE:NEAR, EDITA_BYTE:NEAR
        EXTRN   IMP_PRONTO:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   EDITOR_PRONTO:BYTE
DATA_SEG        ENDS

;-----------------------------------------------------------------------;
; Este e o controlador central. Durante a edicao e apresentacao normal, ;
; esta rotina le caracteres do teclado e, se for uma tecla de           ;
; comando (como tecla de cursor), CONTROLA chama as rotinas             ;
; que fazem o trabalho real. Este controle e feito para as telas        ;
; especiais listadas em TAB_CONTROLE, onde os endereços das             ;
; rotinas sao colocados apos os nomes das teclas.                       ;
;    Se o caractere nao for especial, entao deve ser colocado           ;
; diretamente no buffer do setor -- este e o modo de edicao.            ;
;                                                                       ;
; Usa:      LE_BYTE, EDITA_BYTE, IMP_PRONTO                             ;
; Le:       EDITOR_PRONTO                                               ;
;-----------------------------------------------------------------------;
CONTROLA        PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    DX
LOOP_CONT:
        CALL    LE_BYTE             ;Le caractere em AL
        OR      AH,AH               ;AH = 0 se nenhum caractere lido
                                    ;   -1 se codigo estendido
        JZ      NENHUM_CAR_LIDO     ;Nenhum caractere, tenta de novo
        JS      TECLA_ESPECIAL      ;Le codigo estendido
        MOV     DL,AL
        CALL    EDITA_BYTE          ;Foi caractere normal, edita byte.
        JMP     LOOP_CONT           ;Le outro caractere
```

```
TECLA_ESPECIAL:
        CMP     AL,68           ;F10--saida?
        JE      FIM_CONTROLE    ;Sim, sai
                                ;Usa BX para verificar tabela
        LEA     BX,TAB_CONTROLE
LOOP_ESPECIAL:
        CMP     BYTE PTR [BX],0 ;Fim da tabela?
        JE      FORA_TABELA     ;Sim, tecla nao estava na tabela
        CMP     AL,[BX]         ;Achou o codigo entrado?
        JE      CONTROLE        ;Sim, envia
        ADD     BX,3            ;Nao, tenta proxima entrada
        JMP     LOOP_ESPECIAL   ;Verifica proxima entrada na tabela

CONTROLE:
        INC     BX              ;Aponta endereco na rotina
        CALL    WORT PTR [BX]   ;Chama rotina
        JMP     LOOP_CONT       ;Espera outra tecla

FORA_TABELA:                    ;Somente le o proximo caractere
        JMP     LOOP_CONT

NENHUM_CAR_LIDO:
        LEA     DX,EDITOR_PRONTO
        CALL    IMP_PRONTO      ;Apaga caracteres invalidos digitados
        JMP     LOOP_CONT       ;Tenta de novo
FIM_CONTROLE:
        POP     DX
        POP     BX
        POP     AX
        RET
CONTROLA        ENDP
```

Esse conserto não cria muitos problemas, mas torna CONTROLA um pouco menos elegante. A elegância é uma virtude pela qual devemos lutar. A elegância e a clareza sempre caminham lado a lado, e nossas regras de projeto modular são realizadas mais facilmente com uma maior elegância no programa.

Resumo

CONTROLA é elegante porque é uma solução simples para um problema. Ao invés de usar muitas comparações para cada caractere especial que possamos digitar, montamos uma tabela para fazer a busca. Isso tornou CONTROLA mais simples, e portanto mais confiável, do que um programa contendo instruções diversas para cada condição que poderia surgir. Acrescentando nosso pequeno acerto, tornamos CONTROLA mais complicado — não muito nesse caso, mas certos consertos podem acabar tornando as rotinas muito mais complicadas.

Se você se encontrar realizando muitos reparos em uma rotina, tornando-a demasiado complicada, escreva outras rotinas que possam remover essa complexidade. E sempre verifique as condições de limite tanto antes quanto depois de escrever uma nova rotina a um programa principal. Se fizer assim, irá economizar muitas "latas de inseticida".

Não podemos deixar de enfatizar a importância de se testar as rotinas com as condições de limite, além de se seguir as regras do projeto modular. Ambas as técnicas levam a programas melhores e mais confiáveis. No próximo capítulo, veremos outra forma de se depurar programas.

26

GRAVANDO SETORES MODIFICADOS

Gravando no Disco

Mais Técnicas de Depuração

Fazendo um Mapa da Estrada

Descobrindo Erros

Symdeb

Depuração Simbólica

Troca de Tela

Resumo

Quase já temos um programa Dskpatch utilizável. Neste capítulo, vamos montar a rotina para gravar um setor modificado no disco, e no próximo capítulo, vamos escrever uma rotina para mostrar a segunda metade do setor. Dskpatch ainda não estará pronto, e já dissemos que os programas nunca estão prontos, porém o escopo do que está sendo abordado neste livro estará pronto.

Gravando no Disco

Gravar um setor de volta ao disco pode ser desastroso se não for feito intencionalmente. Todas as funções do Dskpatch dependem das teclas de função F1, F2 e F10, e das teclas do cursor. Mas qualquer uma dessas teclas poderia ser pressionada por acidente. Felizmente, o mesmo não acontece para as teclas de função com o "shift", de modo que usaremos a tecla F5 juntamente com Shift para gravar um setor do disco. Isto evitará que escrevamos um setor de volta ao disco, a menos que realmente queiramos.

Faça as seguintes alterações em CONTROLE.ASM, e acrescente GRAVA_SETOR à tabela:

Listagem 26-1. Alterações em CONTROLE.ASM

```
DATA_SEG        SEGMENT PUBLIC
        EXTRN   LE_SETOR:NEAR                                   ;Em ES_DISCO.ASM
        EXTRN   ANT_SETOR:NEAR                                  ;Em ES_DISCO.ASM
        EXTRN   FANTASMA_CIMA:NEAR, FANTASMA_BAIXO:NEAR         ;Em FANTASMA.ASM
        EXTRN   FANTASMA_ESQ:NEAR, FANTASMA_DIR:NEAR
        EXTRN   GRAVA_SETOR:NEAR                                ;Em ES_DISCO.ASM

;-------------------------------------------------------------------------;
; Esta tabela contem as teclas ASCII estendidas e os enderecos            ;
; das rotinas que devem ser chamadas quadno as teclas forem pressionadas  ;
;       O formato da tabela e:                                            ;
;           DB      60          Codigo estendido para F2                  ;
;           DW                                                            ;
;-------------------------------------------------------------------------;
                LABEL   BYTE
        DB      59                                              ;F1
        DW      OFFSET CGROUP:ANT_SETOR
        DB      60                                              ;F2
        DW      OFFSET CGROUP:PROX_SETOR
        DB      72                                              ;Cursor acima
        DW      OFFSET CGROUP:FANTASMA_CIMA
        DB      80                                              ;Cursor abaixo
        DW      OFFSET CGROUP:FANTASMA_BAIXO
        DB      75                                              ;Cursor esquerda
        DW      OFFSET CGROUP:FANTASMA_ESQ
        DB      77                                              ;Cursor direita
        DW      OFFSET CGROUP:FANTASMA_DIR
        DB      88                                              ;Shift F5
        DW      OFFSET CGROUP:GRAVA_SETOR
        DB      0                                               ;Fim da tabela
DATA_SEG        ENDS
```

GRAVA_SETOR é quase idêntico a LE_SETOR. A única alteração é que queremos gravar, e não ler um setor. Enquanto a função INT 25h pede ao DOS para ler um setor, outra função paralela, INT 26h, pede ao DOS para gravar um setor no disco. Aqui está GRAVA_SETOR. Coloque-a no arquivo ES_DISCO.ASM:

Listagem 26-2. Acrescenta Esta Rotina a ES_DISCO.ASM

```
        PUBLIC  GRAVA_SETOR

;-------------------------------------------------------------------------;
; Esta rotina grava o setor de volta ao disco.                            ;
;                                                                         ;
; Le:       NR_DRIVE, NR_SETOR, SETOR                                     ;
;-------------------------------------------------------------------------;
```

```
GRAVA_SETOR     PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,NR_DRIVE     ;Numero do drive
        MOV     CX,1            ;Grava 1 setor
        MOV
        LEA     BX,SETOR
        INT     26h             ;Grava no disco
        POPF                    ;Descarta informacao de flag
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
GRAVA_SETOR     ENDP
```

Agora monte Controle e Es_disco, mas não tente executar a função de gravação ainda. Apanhe um disco velho com que você não se importe muito e coloque-o no drive A, com o seu disco de programa em algum outro drive, como o B. Execute Dskpatch através do drive B (ou outro que você escolheu), para que o programa leia o primeiro setor do seu disco "cobaia" no drive A. Antes de continuar, tenha certeza de que se o disco no drive A vier a ser destruído, isso não causará pânico e desmaios.

Altere um byte no seu setor e anote o local e o valor do byte que foi alterado. Depois, pressione a tecla F5 juntamente com a Shift. Você verá a luz vermelha do drive acender-se: você acabou de gravar um setor modificado de volta ao drive A.

A seguir, pressione F2 para ler o próximo setor (setor 1), e depois F1 para ler o anterior (seu setor de origem, número 0). Você deverá ver o setor modificado de volta à tela. Recupere o byte anterior desse setor e grave-o de volta no Drive A, para restaurar a integridade do seu "cobaia".

Mais Técnicas de Depuração

O que aconteceria se tivéssemos cometido um pequeno erro em nosso programa? Dskpatch é suficientemente grande para termos problemas usando o Debug para encontrar o erro. Além disso, Dskpatch é composto por nove arquivos diferentes que devemos linkeditar para formar o DSKPATCH.COM. Como podemos encontrar qualquer uma das sub-rotinas nesse programão sem que tenhamos que examinar cuidadosamente o programa inteiro? Como veremos neste capítulo, há dois modos de encontrar rotinas: usando um mapa da estrada, que podemos obter com LINK, ou usando o SYMDEB da Microsoft no lugar do DEBUG.

Quando escrevemos inicialmente Dskpatch, alguma coisa deu errado quando acrescentamos GRAVA_SETOR; ao pressionarmos as teclas Shift+F5 nossa máquina parou. Mas não pudemos encontrar nada de errado em GRAVA_SETOR, e as únicas outras alterações foram na tabela TAB_CONTROLE. Tudo parecia estar correto.

Finalmente, descobrimos que o erro estava em uma definição no controlador. O erro estava na entrada da tabela TAB_CONTROLE para a definição de GRAVA_SETOR. De alguma forma digitamos um DW ao invés de DB na tabela, de modo que o endereço de GRAVA_SETOR ficou um byte defasado na memória. Você pode ver o erro em itálico nessa cópia da tabela:

```
TAB_CONTROLE    LABEL   BYTE
                .
                .
                .
        DB      77                      ;Cursor direita
        DW      OFFSET CGROUP:FANTASMA_DIR
```

```
        DW      88                      ;Shift F5
        DW      OFFSET CGROUP:GRAVA_SETOR
        DB      0                       ;Fim da tabela
DATA_SEG        ENDS
```

Como exercício de depuração, faça essa mudança em seu arquivo CONTROLE.ASM, e depois siga as instruções na próxima seção.

Fazendo um Mapa da Estrada

Vamos aprender como usar LINK para fazer um mapa de Dskpatch. Esse mapa nos ajudará a encontrar as rotinas e as variáveis na memória.

O comando LINK que usamos tornou-se muito longo:

```
LINK DSKPATCH ES_DISCO DISP_SET ES_VIDEO CURSOR CONTROLE ES_TEC FANTASMA EDITOR;
```

e ainda vamos aumentar isso. Isso significa que vamos continuar digitando arquivo após arquivo? Não, há um modo muito mais fácil. O LINK nos permite fornecer um arquivo de *resposta automática* contendo toda a informação. Com esse arquivo, que chamaremos de linkinfo, podemos simplesmente digitar:

```
LINK @LINKINFO
```

e LINK lerá toda essa informação do arquivo.

Com os nomes usados até aqui, linkinfo ficaria assim:

```
DSKPATCH ES_DISCO DISP_SET ES_VIDEO CURSOR +
CONTROLE ES_TEC FANTASMA EDITOR
```

O sinal de mais (+) na primeira linha diz ao LINK para continuar lendo os nomes de arquivo da segunda linha.

Podemos também acrescentar a esse arquivo uma informação que diz ao LINK para criar um mapa das rotinas e variáveis no nosso programa:

```
DSKPATCH ES_DISCO DISP_SET ES_VIDEO CURSOR +
CONTROLE ES_TEC FANTASMA EDITOR
DSKPATCH
DSKPATCH /MAP;
```

As duas últimas linhas são parâmetros novos. O primeiro, *dskpatch*, diz ao LINK que queremos um arquivo .EXE chamado DSKPATCH.EXE; a segunda nova linha (a última) diz ao LINK para criar um arquivo de listagem chamado DSKPATCH.MAP — para criar nosso mapa. A chave */map* diz ao LINK para fornecer uma lista de todas as rotinas e variáveis que declaramos como sendo "public".

Crie um arquivo de mapa linkeditando novamente o Dskpatch com esse arquivo de resposta linkinfo. O arquivo de mapa produzido pelo linkeditor possui cerca de 120 linhas. Isso não dá para ser reproduzido inteiramente aqui, mas vamos apresentar um trecho de maior interesse. Eis uma listagem parcial do arquivo de mapa, DSKPATCH.MAP:

```
Warning: no stack segment

Start     Stop      Length    Name                    Class
00000H    007E5H    007E6H    CODE_SEG
007F0H    0291FH    02130H    DATA_SEG

Origin    Group
0000:0    CGROUP
```

```
 Address          Publics by Name
 0000:0677        BACK_SPACE
 0000:048F        CLEAR_SCREEN
 0000:04D1        CLEAR_TO_END_OF_LINE
 0000:07F2        CURRENT_SECTOR_NO
 0000:04B1        CURSOR_RIGHT
 0000:07F4        DISK_DRIVE_NO
 0000:04F0        DISPATCHER
 0000:01F3        DISP_HALF_SECTOR
                  .
                  .
                  .
 0000:0370        WRITE_HEX_DIGIT
 0000:03DB        WRITE_PATTERN
 0000:06FE        WRITE_PHANTOM
 0000:0440        WRITE_PROMPT_LINE
 0000:013A        WRITE_SECTOR
 0000:0428        WRITE_STRING

 Address          Publics by Value
 0000:0120        LE_SETOR
 0000:013A        GRAVA_SETOR
 0000:0154        ANT_SETOR
 0000:0174        PROX_SETOR
 0000:0190        INIC_DISP_SET
 0000:01F3        DISP_MEIO_SETOR
                  .
                  .
                  .
 0000:07F5        LINHAS_ANTES_SETOR
 0000:07F6        NR_LINHA_CAB
 0000:07F7        CAR_1
 0000:07FD        CAR_2
 0000:080E        NR_LINHA_PRONTO
 0000:080F        EDITOR_PRONTO
 0000:0844        SETOR
 0000:2912        CURSOR_FANT_X
 0000:2913        CURSOR_FANT_Y

Program entry point at 0000:0100
```

Nota: A primeira parte dessa tabela foi, propositadamente, deixada com os nomes originais em inglês. O motivo é que, por se encontrar em ordem alfabética, não seria possível descobrir, pelo texto, o endereço dos PUBLICS para as primeiras rotinas em português. (N.T.)

Há três partes principais nesse *mapa de carga* (ele possui esse nome porque nos diz onde as rotinas são carregadas na memória). A primeira mostra uma lista dos segmentos do nosso programa. Dskpatch possui somente dois segmentos, CODE_SEG e DATA_SEG, que estão agrupados, de modo que você verá esses dois segmentos na lista.

A próxima parte do mapa mostra nossas rotinas e variáveis "public", listadas em ordem alfabética. LINK lista somente as rotinas e variáveis que você declarou como PUBLIC — visíveis ao mundo exterior. Se você está depurando um programa muito grande, pode declarar todas as rotinas e variáveis como public, só para poder encontrá-las nessa lista.

A parte final do mapa mostra todas as rotinas e variáveis novamente, mas dessa vez na ordem em que aparecem na memória.

Ambas as listas incluem o endereço de memória de cada rotina ou variável PUBLIC. Se você verificar essa lista, verá que nossa rotina CONTROLA começa no endereço 4F0h. Usaremos esse endereço agora, para encontrar o erro em Dskpatch.

Descobrindo Erros

Se você estiver com a versão de Dskpatch com o erro, verá que tudo o mais funciona, menos a tecla Shift-F5, que faz com que a máquina pare de funcionar. Você provavelmente não vai querer tentar essa combinação de teclas; ninguém pode dizer o que vai acontecer no seu computador.

Como tudo o mais funcionou (e ainda funciona) com a exceção de Shift-F5, nossa primeira idéia quando escrevemos o programa foi que o erro estaria em GRAVA_SETOR. Para encontrar esse erro, poderíamos começar procurando o erro dentro de GRAVA_SETOR. Ao invés disso, tomamos um rumo diferente.

Sabemos que CONTROLA funciona corretamente, pois tudo mais (as teclas de cursor, F1, F2 e F10) está correto. Isso significa que CONTROLA é um bom ponto de partida para procurar o erro em Dskpatch.

Se você procurar a rotina CONTROLA (no Capítulo 25), verá a seguinte instrução:

```
CALL      WORD PTR [BX]
```

que é o coração de CONTROLA, pois chama todas as outras rotinas. Em particular, essa instrução CALL chamará GRAVA_SETOR quando pressionarmos Shift-F5. Vamos começar nossa procura aqui.

Vamos usar o Debug para executar o Dskpatch com um "breakpoint" nessa instrução. Logicamente, isso significa que precisamos do endereço dessa instrução, e podemos encontrá-lo desassemblando CONTROLA, que se inicia em 4F0h. Após um U 4F0, seguido por outro comando U, você deverá ver o comando CALL:

```
                •
                •
                •
2C14:0517 EBF2          JMP     050B
2C14:0519 43            INC     BX
2C14:051A FF17          CALL    [BX]
2C14:051C EBD5          JMP     04F3
                •
                •
                •
```

Agora que já sabemos que a instrução CALL está na posição 51Ah, podemos colocar o breakpoint nesse endereço, e depois executar o programa passo a passo dentro de GRAVA_SETOR.

Primeiro, usamos o comando G 51A para executar Dskpatch até essa instrução. Você verá que o programa inicia, e espera que você digite um comando. Pressione Shift-F5, pois esse é o comando que está causando problemas. Você verá o seguinte:

```
-G 51A

AX=FF58  BX=28A3  CX=2820  DX=080F  SP=FFF6  BP=419A  SI=03CC  DI=0001
DS=2C14  ES=2C14  SS=2C14  CS=2C14  IP=051A   NV UP DI PL NZ NA PE NC
2C14:051A FF17          CALL    [BX]                          DS:28A3=3A00
```

Nesse ponto, o registrador BX está apontando para uma palavra que deve conter o endereço de GRAVA_SETOR. Vamos ver:

```
-D 28A3 L 2
2C14:28A0                       00  3A
-
```

Em outras palavras, estamos tentando chamar uma rotina localizada em 3A00h (lembre-se que o byte menor é colocado na frente). Mas se olharmos em nosso mapa de memória, podemos ver que GRAVA_SETOR deveria estar em 13Ah. De fato, também podemos dizer através da leitura do mapa que não há qualquer rotina em 3A00h. O endereço está totalmente errado!

Em nossa procura inicial, quando escrevemos o programa, não foi difícil descobrir o erro. Sabíamos que a rotina CONTROLA e a tabela estavam em concordância, pois todas as outras funções estavam funcionando corretamente, de modo que olhamos mais atentamente para os dados do Shift-F5, e descobrimos um DW onde deveria haver um DB. Um mapa do caminho torna a depuração muito mais simples. Agora vamos dar uma olhada no Symdeb.

Symdeb

Symdeb (*Symbolic Debugging — Depuração Simbólica*) é um programa que a Microsoft inclui com a versão 3.00 do seu pacote de macro assembler. Como você pode ver nessa seção, ele é tão importante que, se você não o tiver, possivelmente irá pensar em atualizar o seu macro assembler.

Como tanto o Debug quanto o Symdeb foram escritos pela Microsoft, o Symdeb compartilha da maioria (se não todos) dos comandos do Debug. Isso também inclui uma série de comandos muito úteis que você não encontrará no Debug, *além de* incluir algumas características que valem o seu peso em ouro. Usaremos duas dessas características nesse capítulo: depuração simbólica e troca de tela.

Depuração Simbólica

A depuração simbólica, que dá nome ao Symdeb, nos permite ver os nomes das rotinas e variáveis, ao invés do endereço, em nossas listagens desassembladas (U). Por exemplo, se usarmos o Debug para desassemblar a primeira linha do Dskpatch, vemos:

```
2C14:0100  E88C03      CALL    046F
```

Com o Symdeb, por outro lado, temos o seguinte:

```
3245:0100  E88C03      CALL    LIMPA_TELA
```

Qual desses é mais fácil de se ler?

Troca de Tela

A segunda nova característica, troca de tela, é prática para a depuração de Dskpatch. O programa salta por toda a tela, escrevendo em lugares diferentes. Na seção anterior, quando usamos o Debug, o Debug começou a escrever na tela e eventualmente perdemos a tela do Dskpatch.

O Symdeb, no entanto, mantém duas telas separadas: uma para o Dskpatch e outra para ele mesmo. Sempre que o Dskpatch estiver ativo, vemos a sua tela; sempre que Symdeb estiver ativo, vemos a tela *dele*. Vamos ver alguns exemplos para ter uma idéia mais clara de como isso funciona.

Antes que possamos usar essa característica do Symdeb, precisamos criar um arquivo de símbolo com um programa chamado Mapsym. Mapsym lê o arquivo .MAP que havíamos criado neste capítulo e converte-o em um arquivo de símbolo:

```
A>MAPSYM DSKPATCH
Microsoft (R) Symbol File Generator   Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.

     Program entry point at 0000:0100
```

Nesse caso, Mapsym criou um arquivo de símbolo chamado DSKPATCH.SYM. Depois, iniciamos o Symdeb com o arquivo de símbolo e o arquivo .COM:

```
A>SYMDEB /S DSKPATCH.SYM DSKPATCH.COM
Microsoft (R) Symbolic Debug Utility   Version 4.00
Copyright (C) Microsoft Corp 1984, 1985. All rights reserved.

Processor is [8086]
```

A chave /S em nosso comando diz ao Symdeb para usar sua característica de troca de tela. Ele não usa essa característica se não indicarmos, pois torna o Symdeb bem mais lento.

Antes de executarmos novamente a seção de depuração anterior, vamos dar uma rápida olhada no início de Dskpatch:

```
-U
330E:0100 E88C03        CALL    CLEAR_SCREEN
330E:0103 E8F402        CALL    WRITE_HEADER
330E:0106 E81700        CALL    READ_SECTOR
330E:0109 E88400        CALL    INIC_SEC_DISP
330E:010C 8D160F08      LEA     DX,[EDITOR_PROMPT]
330E:0110 E82D03        CALL    WRITE_PROMPT_LINE
330E:0113 E8DA03        CALL    DISPATCHER
330E:0116 CD20          INT     20
-
```

Você pode ver a bela apresentação dos nomes (ao invés dos endereços) que o Symdeb realiza.

Quando desassemblamos CONTROLA para descobrirmos o endereço do CALL WORD PTR [BX], tivemos primeiro que olhar no mapa para descobrir o endereço da rotina, e depois digitar U 4F0 para desassemblá-la. Com o Symdeb, as coisas são muito mais simples: podemos apenas digitar U CONTROLA para desassemblar nossa rotina:

```
-U CONTROLA
CGROUP:CONTROLA:
330E:04F0 50            PUSH    AX
330E:04F1 53            PUSH    BX
330E:04F2 52            PUSH    DX
330E:04F3 E80401        CALL    LE_BYTE
330E:04F6 0AE4          OR      AH,AH
330E:04F8 7426          JZ      CONTROLA+30 (0520)
330E:04FA 7807          JS      CONTROLA+30 (0503)
```

Após mais dois comandos U, encontramos nossa instrução CALL:

```
330E:0514 83C303        ADD     BX,+03
330E:0517 EBF2          JMP     CONTROLA+1B (050B)
330E:0519 43            INC     BX
330E:051A FF17          CALL    [BX]
330E:051C EBD5          JMP     CONTROLA+03 (04F3)
```

Digite G 51A, como antes, e depois pressione Shift-F5. Se você possui o Symdeb, verá o Dskpatch desenhar sua tela. Depois você retornará ao Symdeb após pressionar Shift-F5. Desta vez, entretanto, você não verá a tela do Dskpatch, pois o Symdeb trocou as telas. Para voltar à tela do Dskpatch, pressione a tecla de barra invertida (\ ) e a tecla Enter. Quando a tela do Dskpatch aparecer, pressione qualquer tecla para voltar novamente ao Symdeb.

Há um ponto sutil sobre o Symdeb que você deve ter notado aqui. Se olharmos as instruções desassembladas, veremos algo assim:

```
330E:051C   EBD5        JMP     CONTROLA+03 (04F3)
```

ao invés de:

```
330E:051C   EBD5        JMP     LOOP_CONT
```

Por que o Symdeb não usou o rótulo LOOP_CONT? Não definimos os rótulos nessa rotina como sendo PUBLIC. Se voltássemos e definíssemos todos os rótulos como PUBLIC em nosso programa, a listagem desassemblada apresentaria todos os seus nomes. (Se você fizer isso, lembre-se de reconstruir o arquivo de símbolo com o Mapsym.)

Resumo

Isso termina nossa discussão sobre técnicas de depuração. Só temos mais três capítulos neste livro. No capítulo seguinte, colocaremos rotinas para trocar as duas metades do setor na tela. Depois, nos dois capítulos finais, aprenderemos mais sobre as diferenças entre os arquivos .COM e .EXE, e daremos uma olhada final no comando ASSUME e superposições de segmento.

A propósito: não se esqueça de retirar o "bug" que colocamos na TAB_CONTROLE. Uma pequena dose de RAID eliminará esse incômodo "inseto".

27

O OUTRO MEIO SETOR

Rolando Meio Setor

Resumo

O ideal seria que nosso programa se comportasse como um processador de texto quando tentasse mover o cursor abaixo da última linha: a tela moveria uma linha acima, com uma nova linha aparecendo embaixo. Não iremos fazer algo tão sofisticado aqui. Neste capítulo, incluiremos as versões básicas dessas duas rotinas, TELA_SOBE e TELA_DESCE, que fazem o movimento dentro do setor, passando de uma metade para a outra, e não somente uma ou algumas linhas.

Rolando Meio Setor

Nossas versões antigas de FANTASMA_CIMA e FANTASMA_BAIXO deixam o cursor no alto da tela do meio-setor sempre que tentamos mover o cursor para uma linha acima do topo; o mesmo acontece para a última linha. Vamos alterar FANTASMA_CIMA e FANTASMA_BAIXO para que possamos chamar TELA_SOBE ou TELA_DESCE quando o cursor tentar mover para fora dos limites verticais da tela. Essas duas novas rotinas trocarão a tela e colocarão o cursor em sua nova posição.

Eis as versões modificadas de FANTASMA_CIMA e FANTASMA_BAIXO (em FANTASMA.ASM):

Listagem 27-1. Alterações em FANTASMA.ASM

```
FANTASMA_CIMA  PROC      NEAR
          CALL     APAGA_FANTASMA      ;Apaga na posicao atual
          DEC      CURSOR_FANT_Y       ;Sobe cursor uma linha
          JNS      NAO_ESTAVA_TOPO     ;Nao estava no alto, imprime
          MOV      CURSOR_FANT_Y,0     ;Estava no alto, coloca de volta
          CALL     TELA_DESCE          ;Estava no alto, desce tela
NAO_ESTAVA_TOPO:
          CALL     IMP_FANTASMA        ;Imprime o cursor na posicao
          RET
FANTASMA_CIMA  ENDP

FANTASMA_BAIXO PROC      NEAR
          CALL     APAGA_FANTASMA      ;Apaga na posicao atual

          INT      CURSOR_FANT_Y       ;Abaixa cursor uma linha
          CMP      CURSOR_FANT_Y,16    ;Estava no fundo?
          JB       NAO_ESTAVA_FUNDO    ;Nao, imprime fantasma
          MOV      CURSOR_FANT_Y,15    ;Sim, coloca de volta
          CALL     TELA_SOBE           ;Sim, sobe a tela
NAO_ESTAVA_FUNDO:
          CALL     IMP_FANTASMA        ;Imprime o cursor fantasma
          RET
FANTASMA_BAIXO ENDP
```

Não se esqueça de alterar o cabeçalho para FANTASMA_CIMA e FANTASMA_BAIXO, de modo que mencione que essas duas rotinas agora usam TELA_SOBE e TELA_DESCE:

Listagem 27-2. Alterações em FANTASMA.ASM

```
;-----------------------------------------------------------------------;
; Estas quatro rotinas movem o cursor fantasma.                         ;
;                                                                       ;
; Usa:          APAGA_FANTASMA, IMP_FANTASMA                            ;
;               TELA_DESCE, TELA_SOBE                                   ;
; Le:           CURSOR_FANT_X, CURSOR_FANT_Y                            ;
; Escreve:      CURSOR_FANT_X, CURSOR_FANT_Y                            ;
;-----------------------------------------------------------------------;
```

TELA_SOBE e TELA_DESCE são rotinas muito simples, pois elas simplesmente trocam o vídeo para a outra metade do setor. Por exemplo, se estivermos vendo a primeira metade, e FANTASMA_BAIXO chamar TELA_SOBE, veremos a segunda metade do setor. A rotina TELA_SOBE altera DESL_SETOR para 256, o início do segundo meio setor, move o cursor para o início

da tela, imprime a tela para a segunda metade, e finalmente imprime o cursor fantasma no topo da tela.

Você pode ver todos os detalhes de TELA_SOBE e TELA_DESCE na listagem a seguir. Acrescente-a em FANTASMA.ASM.

Listagem 27-3. Acrescente Estas Rotinas em FANTASMA.ASM

```
        EXTRN   DISP_MEIO_SETOR:NEAR, GOTO_XY:NEAR
DATA_SEG
                SEGMENT PUBLIC
        EXTRN   DESL_SETOR:WORD
        EXTRN   LINHAS_ANTES_SETOR:BYTE
                ENDS

;-------------------------------------------------------------------------------
; Estas duas rotinas fazem a troca entre as duas metades do setor.             ;
;                                                                              ;
; Usa:      IMP_FANTASMA, DISP_MEIO_SETOR, APAGA_FANTASMA, GOTO_XY             ;
;           SALVA_CURSOR_REAL, RESTAURA_CURSOR_REAL                            ;
; Le:       LINHAS_ANTES_SETOR                                                 ;
; Escreve:  DESL_SETOR, CURSOR_FANT_Y                                          ;
;-------------------------------------------------------------------------------
TELA_SOBE       PROC    NEAR
        PUSH    DX
        CALL    APAGA_FANTASMA          ;Remove o cursor fantasma
        CALL    SALVA_CURSOR_REAL       ;Salva posicao do cursor real
        XOR     DL,DL                   ;Define cursor para tela de meio-setor
        MOV     DH,LINHAS_ANTES_SETOR
        ADD     DH,2
        CALL    GOTO_XY
        MOV     DX,256                  ;Apresenta a segunda metade do cursor
        MOV     DESL_SETOR,DX
        CALL    DISP_MEIO_SETOR
        CALL    RESTAURA_CURSOR_REAL    ;Restaura posicao do cursor real
        MOV     CURSOR_FANT_Y,0         ;Cursor no alto da segunda metade
        CALL    IMP_FANTASMA            ;Restaura o cursor fantasma
        POP     DX
        RET
TELA_SOBE       ENDP

TELA_DESCE      PROC    NEAR
        PUSH    DX
        CALL    APAGA_FANTASMA          ;Remove o cursor fantasma
        CALL    SALVA_CURSOR_REAL       ;Salva posicao do cursor
        XOR     DL,DL                   ;Define cursor para tela de meio-setor
        MOV     DH,LINHAS_ANTES_SETOR
        ADD     DH,2
        CALL    GOTO_XY
        XOR     DX,DX                   ;Apresenta a primeira metade do cursor
        MOV     DESL_SETOR,DX
        CALL    DISP_MEIO_SETOR
        CALL    RESTAURA_CURSOR_REAL    ;Restaura posicao do cursor real
        MOV     CURSOR_FANT_Y,15        ;Cursor no fundo da primeira metade
        CALL    IMP_FANTASMA            ;Restaura o cursor fantasma
        POP     DX
        RET
TELA_DESCE      ENDP
```

TELA_SOBE e TELA_DESCE trabalham muito bem, embora haja um pequeno probleminha com eles pelo modo com que Dskpatch se encontra agora. Inicie o Dskpatch e deixe o cursor no alto da tela. Pressione a tecla para subir o cursor, e você verá que o programa vai reimprimir a primeira metade do setor. Por quê? Não verificamos essa condição de limite. Dskpatch sempre irá reimprimir a tela, quando você tentar subir acima da primeira metade do setor.

Agora um desafio para você: modifique Dskpatch para que ele verifique ambos os limites. Se você estiver na primeira linha da primeira metade, e pressionar a tecla para subir uma linha, nada deve ser feito. Da mesma forma, se você estiver na linha de baixo do segundo setor, e pressionar a tecla de cursor para descer uma linha, novamente isso deverá ser ignorado.

Resumo

Isso termina com nosso trabalho com o Dskpatch neste livro. Nossa intenção foi usar o programa Dskpatch como um exemplo "vivo" da evolução de um programa em linguagem assembly, ao mesmo tempo dando a você um programa útil, e um grupo de rotinas práticas para os seus novos projetos de programação. Mas o programa que desenvolvemos aqui não está tão completo quanto poderia estar. Além disso, você tem a possibilidade de adquirir um disco com todos os passos que realizamos até aqui, até a sua versão final e melhorada. E você poderá até mesmo alterar a versão do disco, pois "um programa nunca está completo ... mas há uma hora e que temos que despachá-lo para os usuários".

Vamos terminar este livro com uma mudança de passo. Nos dois capítulos seguintes, passaremos a dois assuntos avançados: relocação e mais sobre segmentos.

PARTE IV

DIVERSOS

28

RELOCAÇÃO

Segmentos Múltiplos

Relocação

Programas .COM versus .EXE

Um assunto que parece estar sempre rodeado de mistério é a diferença entre os arquivos .COM e .EXE e o significado dos programas relocáveis. Como parte de nossa mudança de passo nestes dois últimos capítulos, vamos ver algo sobre relocação e como montar programas com mais de 64K — e não o que você necessariamente vai querer, mas o que muitas pessoas querem.

Segmentos Múltiplos

Logo que começarmos a escrever programas com mais de 64K, vamos nos encontrar em dificuldades com os arquivos .COM. Por quê? É o que vamos descobrir aqui.

Primeiro de tudo, cada programa deve ser montado dentro de um ou mais segmentos, com um máximo de 64K em cada um. Mas muitos programas ampliam o seu uso da memória usando diversos segmentos diferentes; por exemplo, um segmento de código para o programa, um segmento de dados para os dados, um segmento de pilha para a pilha e dados temporários. Se cada um desses três segmentos for totalmente utilizado, completaríamos 3 X 64K = 192K de memória. É assim que temos acesso a mais memória, e é aí que aparece a diferença entre programas .EXE e .COM: programas .EXE são feitos especificamente para esse tipo de trabalho.

Todos os nossos programas neste livro são programas .COM, com um segmento ou um grupo. Lembre-se de que a pseudo-op GROUP simplesmente combina diversos segmentos em uma única unidade que atua como um segmento. Se quiséssemos usar mais de um segmento para lidar com mais de 64K de memória, teríamos algum trabalho adicional. Vejamos um exemplo.

Nosso programa para imprimir um string de caracteres no Capítulo 3 serve muito bem. Esse exemplo, escrito com grupos em linguagem assembly, ficaria assim:

```
GRUPO     GROUP    SEG_COD, SEG_DADOS
          ASSUME   CS:GRUPO, DS:GRUPO
SEG_COD            SEGMENT PUBLIC
          ORG      100h
IMP_STRING         PROC     FAR
          MOV      AH,9                        ;Chamada para saida de string
          MOV      DX,OFFSET GRUPO:STRING      ;Carrega endereco do string
          INT      21h                         ;Imprime string
          INT      20h                         ;Retorna ao DOS
IMP_STRING         ENDP

SEG_COD            ENDS

SEG_DADOS          SEGMENT PUBLIC
STRING    DB       "Ola, DOS aqui.$"
SEG_DADOS          ENDS

          END      IMP_STRING
```

Os dois segmentos SEG_COD e SEG_DADOS são colocados em um único grupo de 64K, CGROUP, de modo que OFFSET GRUPO:STRING dá o deslocamento de STRING desde o início do grupo chamado CGRUPO.

Quando o DOS carrega um programa .COM para a memória, ele atribui todos os quatro registradores de segmento (CS, DS, ES e SS) para o início de CGRUPO, e por isso DS:OFFSET CGRUPO:STRING é o endereço completo de STRING. E se tivéssemos dois segmentos diferentes e nenhum grupo? Não teríamos um limite de 64K para dois segmentos: seria 128K. Como podemos fazer com que os registradores de segmento apontem para seus respectivos segmentos? Usando um programa .EXE, que nos permite usar diversos segmentos, todos começando em diferentes endereços.

O DOS nos permite definir os registradores de segmento para um programa .EXE com a ajuda de algumas instruções do assembler. Essas atribuições não são tão simples como pode parecer, mas voltaremos a isso depois. Primeiro, vamos refazer IMP_STRING como um programa .EXE.

Devemos ter pelo menos dois segmentos para qualquer programa .EXE: o segmento de código e o segmento de pilha. Esses dois segmentos são casos especiais para o DOS. O DOS atribui endereços para os quatro registradores de segmento quando carrega um programa .EXE na memória. O DOS faz com que o registrador CS:IP aponte para a primeira instrução cujo endereço aparece após a pseudo-op END. Em um programa .EXE, esta primeira instrução pode estar em qualquer lugar, enquanto em um programa .COM, esta instrução *deve* ser a primeira instrução no segmento de código.

De forma semelhante, SS:IP aponta para o final da região da pilha definida com a pseudo-op SEGMENT STACK. Por exemplo, essa versão de IMP_STRING possui uma pilha com 80 bytes de comprimento, de modo que IP será setado para 80 — o final da região da pilha dentro do segmento de pilha. Eis o programa:

```
SEG_COD         SEGMENT PUBLIC
IMP_STRING      PROC    FAR
        MOV     AX,SEG_DADOS    ;Endereco de segmento para SEG_DADOS
        MOV     DS,AX           ;Define registrador DS para SEG_DADOS
        MOV     AH,9            ;Chamada para saida de string
        MOV     DX,OFFSET STRING ;Carrega endereco do string
        INT     21h             ;Imprime string

        PUSH    ES              ;Salva endereco de retorno para RET longo, abaixo
        XOR     AX,AX           ;Ha uma instrucao 20h em ES:0

        RET                     ;Retorna ao DOS
IMP_STRING      ENDP

SEG_COD         ENDS

SEG_DADOS       SEGMENT PUBLIC
STRING  DB      'Ola, DOS aqui.$'
SEG_DADOS       ENDS

SEG_PILHA       SEGMENT STACK
        DB      10 DUP ('PILHA ')       ;'PILHA' seguido por 3 espacos
SEG_PILHA       ENDS

        END     IMP_STRING
```

Esse programa estará pronto para ser executado após a linkedição, mas primeiro apague IMPSTR.COM. Se você tiver duas versões de um arquivo, uma com extensão .COM e uma com a extensão .EXE, o DOS executará o arquivo .COM.

Há uma série de diferenças entre esse arquivo .EXE e o nosso arquivo original .COM. No lugar da instrução INT 20h para retornar ao DOS, temos agora diversas instruções "camufladas", começando com PUSH ES. As duas instruções PUSH colocam um endereço longo de retorno na pilha, ES:0. Esse é o endereço do primeiro byte da área de dados de 256 bytes que o DOS define na memória antes do nosso programa, e a primeira instrução nessa área de dados é uma instrução INT 20h.

O registrador CS deve estar apontando para o início dessa área quando executarmos a instrução INT 20h. Foi isso que aconteceu com nosso programa .COM, bem no começo. Mas nosso programa .EXE começa com o registrador CS apontando para o início do segmento de código, e não para a área de dados. Fazendo um FAR RET para ES:0, passamos CS para o início da área de dados e, como você pode ver, ES:0 contém a instrução INT 20h:

```
A>DEBUG IMPSTR.EXE
-U ES:0
39AF:0000 CD20          INT     20
39AF:0002 006000        ADD     [BX+SI+D0],AH
        •
        •
        •
```

A pseudo-op GROUP não está presente, pois agora temos três segmentos diferentes que não estão confinados a uma área total de no máximo 64K. Cada um desses três segmentos é independente, e cada um dos registradores de segmento (CS, DS e SS) aponta para um segmento diferente. CS e SS são setados por meio do DOS, como podemos ver com a ajuda do Debug:

```
A>DEBUG IMPSTR.EXE
-R
AX=0000  BX=0000  CX=0100  DX=0000  SP=0050  BP=0000  SI=0000  DI=0000
DS=39AF  ES=39AF  SS=39C3  CS=39BF  IP=0000   NV UP DI PL NZ NA PO NC
39BF:0000 B8C139        MOV     AX,39C1
```

DS e ES apontam para um segmento mais baixo na memória. Como você pode ver no Capítulo 11, tanto DS quanto ES apontam para a área de dados, de 256 bytes, colocada pelo DOS antes do nosso programa. No programa .COM, reservamos essa área com um comando ORG 100h. Para arquivos .EXE, não precisamos fazer o mesmo, pois os segmentos de código e dados estão em partes diferentes da memória. O segmento de dados está em algum outro lugar, mas o DOS não está apontando para SEG_DADOS. Esta é a razão para a primeira instrução de IMP_STRING. A instrução MOV AX,SEG_DADOS move o número do segmento de SEG_DADOS para o registrador AX. Se olharmos nosso programa na memória:

```
-U
39BF:0000 B8C139      MOV     AX,39C1
39BF:0003 8ED8        MOV     DS,AX
39BF:0005 B409        MOV     AH,09
39BF:0007 BA0000      MOV     DX,0000
39BF:000A CD21        INT     21
39BF:000C 06          PUSH    ES
39BF:000D 33C0        XOR     AX,AX
39BF:000F 50          PUSH    AX
39BF:0010 CB          RETF
39BF:0011 0000        ADD     [BX+SI],AL
        .
        .
        .
```

veremos que essa instrução MOV foi traduzida por MOV AX,39C1, onde 39C1 é o número do segmento SEG_DADOS. Precisamos de duas instruções MOV para colocar esse número no registrador DS, pois não podemos mover um número diretamente para um registrador de segmento qualquer. (Veja a tabela de modos de endereçamento no Apêndice E.)

De onde vem o 39C1? Certamente, nem o assembler nem o linkeditor poderiam saber de antemão onde o DOS carregaria esse programa; só o DOS sabe isso. De fato, é o DOS que atribui esse número, e o processo de cálculo de tais números é conhecido como *relocação*. O DOS faz os cálculos de relocação para programas .EXE, mas não para programas .COM. É por essa razão que os programas .COM são carregados mais rapidamente na memória. Eles também são mais compactos, pois não contêm a informação especial que o DOS usa para realizar os cálculos de relocação.

Por questão de curiosidade, vejamos o que acontece se tentarmos converter nosso programa .EXE em um programa .COM usando o Exe2bin:

```
A>EXE2BIN IMPSTR IMPSTR.COM
File cannot be converted
A>
```

Exe2bin sabe que não pode criar um programa .COM com o nosso arquivo, mas ele não nos diz por quê. Ele nos deixa descobrir por nós mesmos. Vamos dar uma olhada no problema.

O DOS carrega um programa .COM diretamente na memória após criar a área inicial de 256 bytes. Se quisermos segmentos diferentes, com em IMP_STRING, e quisermos criar um arquivo .COM, temos que fazer a relocação por nós mesmos, com instruções em nosso programa. Não é muito difícil, e mostraremos como isso é feito, para que você conheça melhor o modo com que o DOS

reloca os programas. Se você precisar escrever um programa .COM grande, que precise de mais do que 64K de memória, essa técnica será muito útil.

Relocação

Nosso objetivo é setar o registrador DS para o início de SEG_DADOS, e o registrador SS para o início do SEG_PILHA. Podemos fazer isso com alguns truques. Primeiro, temos que nos assegurar de que nossos três segmentos são carregados para a memória na ordem exata:

Segmento de Código
Segmento de Dados
Segmento de Pilha

Felizmente, já cuidamos disso. O linkeditor carrega estes três segmentos na ordem em que aparecem no nosso arquivo. No entanto há um detalhe: quando você usar a técnica a seguir em um arquivo .COM para setar os registradores de segmento, tenha certeza da ordem em que LINK irá carregar os seus segmentos.

Com calculamos o valor de DS? Vamos começar olhando os três rótulos que colocamos nos vários segmentos da listagem a seguir. Esses rótulos são FIM_DO_SEG_COD, FIM_DO_SEG_DADOS e FIM_DO_SEG_PILHA. Eles não estão exatamente onde você poderia estar esperando. Por que não? Bem, quando definimos um segmento como:

```
SEG_COD     SEGMENT     PUBLIC
```

não dizemos realmente ao linkeditor como juntar os vários segmentos. Assim, ele começa cada novo segmento em um limite de parágrafo — em um endereço hexa que termine com zero, como 32C40h. Como o linkeditor salta para o próximo limite de parágrafo para começar um novo segmento, geralmente haverá uma pequena área em branco entre os segmentos. Colocando o rótulo FIM_DO_SEG_COD no início de SEG_DADOS, nós incluimos essa área em branco. Se tivéssemos colocado FIM_DO_SEG_COD no final de SEG_COD, não incluiríamos essa área em branco entre os segmentos. (Veja a listagem desassemblada do programa após a listagem do código fonte. Você verá uma área em branco com zeros ocupando 15 bytes.)

Conforme o valor do registrador DS, SEG_DADOS começa em 39AF:0130, ou 39C2:0000. A instrução OFFSET SEG_COD:FIM_DO_SEG_COD retornará 130h, que é o número de bytes usados por SEG_COD. Divida esse número por 16 e terá o número que precisamos somar ao DOS para que ele aponte para SEG_DADOS. Usamos a mesma técnica para SS.

Eis a listagem do nosso programa, incluindo as instruções de relocação necessárias para um arquivo .COM:

```
            ASSUME   CS:SEG_COD, DS:SEG_DADOS, SS:SEG_PILHA

SEG_COD     SEGMENT PUBLIC
            ORG     100h                ;Reserva area de dados para programa .COM
IMP_STRING  PROC    FAR
            MOV     AX,OFFSET SEG_COD:FIM_DO_SEG_COD
            MOV     CL,4                ;Calcula numero de paragrafos
            SHR     AX,CL               ;(16 bytes) usados pelo seg. de codigo
            MOV     BX,CS
            ADD     AX,BX               ;Soma CS a isto
            MOV     DS,AX               ;Define registrador DS para SEG_DADOS

            MOV     BX,OFFSET SEG_DADOS:FIM_DO_SEG_DADOS
            SHR     BX,CL               ;Calcula parag. usados pelo seg. de dados
            ADD     AX,BX               ;Soma ao valor usado para seg. dados
            MOV     SS,AX               ;Define SS para seg. pilha
            MOV     AX,OFFSET SEG_PILHA:FIM_DO_SEG_PILHA
            MOV     SP,AX               ;Define SP para final de area de pilha
```

```
        MOV     AH,9                    ;Chamada para saida de string
        MOV     DX,OFFSET STRING        ;Carrega endereco do string
        INT     21h                     ;Imprime string

        PUSH    ES                      ;Salva endereco de retorno para RET longo, abaixo
        XOR     AX,AX                   ;Ha uma instrucao INT 20h em ES:0
        PUSH    AX
        RET                             ;Retorna ao DOS
IMP_STRING      ENDP

SEG_COD         ENDS

SEG_DADOS
                SEGMENT PUBLIC
FIM_DO_SEG_COD
                        LABEL   BYTE    FIM_DO_SEG_COD
STRING  DB      "Ola, DOS aqui.$"
SEG_DADOS       ENDS

SEG_PILHA       SEGMENT         PUBLIC
FIM_DO_SEG_DADOS        LABEL   BYTE
        DB      10 DUP ('PILHA  ')      ;'PILHA' seguido por espacos
FIM_DO_SEG_PILHA        LABEL   BYTE
SEG_PILHA       ENDS
        END     IMP_STRING
```

Você pode ver os resultados de todo esse trabalho na seguinte sessão de depuração:

```
A>DEBUG IMPSTR.COM
-U
39AF:0100 B83001    MOV     AX,0130
39AF:0103 B104      MOV     CL,04
39AF:0105 D3E8      SHR     AX,CL
39AF:0107 8CCB      MOV     BX,CS
39AF:0109 03C3      ADD     AX,BX
39AF:010B 8ED8      MOV     DS,AX
39AF:010D BB2000    MOV     BX,0020
39AF:0110 D3EB      SHR     BX,CL
39AF:0112 03C3      ADD     AX,BX
39AF:0114 8ED0      MOV     SS,AX
39AF:0116 B85000    MOV     AX,0050
39AF:0119 8BE0      MOV     SP,AX
39AF:011B B409      MOV     AH,09
39AF:011D BA0000    MOV     DX,0000
-U
39AF:0120 CD21      INT     21
39AF:0122 06                ES
39AF:0123 33C0      XOR     AX,AX
39AF:0125 50
                    RETF
39AF:0127 0000      ADD     [BX+SI],AL
39AF:0129 0000      ADD     [BX+SI],AL
39AF:012B 0000      ADD     [BX+SI],AL
39AF:012D 0000      ADD     [BX+SI],AL
39AF:012F 004865    ADD     [BX+SI+65],CL
39AF:0132 6C        DB      6C
39AF:0133 6C        DB      6C
39AF:0134 6F        DB      6F
39AF:0135 2C20      SUB     AL,20
39AF:0137 44        INC     SP
39AF:0138 4F        DEC     DI
39AF:0139 53
                    AND     [BX+SI+65],CH
39AF:013D 7265      JB      01A4
39AF:013F 2E        CS:
39AF:0140 2400      AND     AL,00
-G 120

AX=0950  BX=0002  CX=0004  DX=0000  SP=0050  BP=0000  SI=0000  DI=0000
DS=39C2  ES=39AF  SS=39C4  CS=39AF  IP=0120     NV UP DI  PL NZ NA PO NC
39AF:0120 CD21              INT     21
```

Fazendo a relocação para mais de um segmento por nós mesmos, aumentamos a quantidade de memória que o programa .COM pode usar. A maioria das pessoas nunca precisa desses truques, mas sabendo como funciona a relocação, podemos entender melhor como o DOS faz a relocação com arquivos .EXE.

Programas .COM versus .EXE

Vamos terminar este capítulo resumindo a diferença entre arquivos .COM e .EXE.

Um arquivo .COM armazenado no disco é essencialmente uma imagem do programa na memória. Por causa disso, um programa .COM é restrito a um único segmento, a não ser que faça sua própria relocação, como fizemos nesse capítulo. Essa restrição de um segmento se deve ao fato de que, em um programa .COM, todos os endereços são indicados por meio de deslocamentos (em uma palavra), que nunca ultrapassam os 64K de um segmento.

Um programa .EXE por outro lado, deixa que o DOS cuide da relocação. Esta delegação de poderes torna o uso de segmentos múltiplos muito fácil nos programas .EXE. Por essa razão, a maioria dos grandes programas são .EXE, ao invés de .COM.

Para uma visão final sobre programas .COM versus .EXE, vejamos mais de perto como o DOS carrega e inicia cada um deles. Isso deve tornar mais clara e concreta a diferença entre os dois tipos de programa. Vamos começar com programas .COM.

Quando o DOS carrega um programa .COM na memória, ele segue os seguintes passos:

- Primeiro, o DOS cria um prefixo de segmento de programa (PSP), que é a área de 256 bytes indicada no Capítulo 11. Entre outras coisas, o PSP contém a linha do comando digitado.
- Depois o DOS copia o arquivo .COM inteiro do disco para a memória, logo após os 256 bytes do PSP.
- A seguir, o DOS preenche todos os registradores de segmento (CS, DS, ES e SS) com o início do PSP.
- Finalmente, O DOS seta o registrador IP para 100h (que é o início do programa .COM) e seta SP para o final do segmento — geralmente FFFE, que é a última palavra dentro do segmento.

Em contraste a esse processo, os passos envolvidos para a carga de um arquivo .EXE são um tanto complicados, pois o DOS faz a relocação. Onde o DOS encontra a informação necessária para a relocação?

Acontece que todo arquivo .EXE possui um cabeçalho que é colocado no início do arquivo. Este cabeçalho, ou *tabela de relocação*, tem sempre pelo menos 512 bytes, e contém toda a informação que o DOS precisa para fazer a relocação. Com as versões mais recentes do macro assembler, a Microsoft incluiu um programa chamado EXEMOD, que podemos usar para ver alguma informação dessa tabela:

```
A>EXEMOD IMPSTR
Microsoft (R) EXE File Header Utility  Version 4.00
Copyright (C) Microsoft Corp. 1985. All rights reserved.

IMPSTR                              (hex)          (dec)
.EXE size (bytes)                     290            656
Minimum load size (bytes)              90            144
Overlay number                          0              0
Initial CS:IP                   0000:0000
Initial SS:SP                   0004:0050             80
Minimum allocation (para)               0              0
Maximum allocation (para)            FFFF          65535
Reader size (para)                     20             32
Relocation table offset                1E             30
Relocation entries                      1

A>
```

No final dessa tabela, você pode ver que temos uma única entrada de relocação para o nosso programa IMPSTR. Toda vez que fizermos referência a um endereço de segmento (como fizemos na instrução MOV AX,SEG_DADOS), o LINK acrescentará uma entrada de relocação nessa tabela. O endereço do segmento não é conhecido até que o DOS carregue nosso programa para a memória, para que possa fornecer o número do segmento.

Há também algumas outras partes interessantes na tabela; por exemplo, os valores iniciais de CS:IP e SS:SP. Esses pares nos dizem os valores iniciais para IP e SP. A tabela também diz ao DOS quanta memória o programa precisa antes que possa ser executado: o tamanho mínimo para carga — *Minimum load size* na tabela.

Como o DOS usa essa tabela de relocação para fornecer os endereços absolutos para posições tais como endereços de segmento, há alguns passos adicionais na carga de um programa .EXE:

- O DOS cria o prefixo de segmento de programa (PSP), assim como ele faz para um programa .COM.
- Segundo, o DOS verifica o cabeçalho .EXE para descobrir onde termina o cabeçalho e começa o programa. Depois ele carrega o restante do programa para a memória após o PSP.
- A seguir, usando a informação no cabeçalho, o DOS descobre e altera todas as referências que devem ser relocadas no programa, como referências a endereços de segmento.
- Depois o DOS seta os registradores ES e DS para que apontem para o início do PSP. Se o programa possui um segmento de dados próprio, ele precisa alterar DS e/ou ES de modo que apontem para o seu segmento de dados.
- Finalmente, o DOS faz com que o registrador CS aponte para o início do segmento de código, com IP definido pela informação no cabeçalho do arquivo .EXE. Da mesma forma, ele seta SS:SP de acordo com a informação da tabela. No caso ilustrado, o cabeçalho diz que SS:SP será colocado me 0004:0050. Isso significa que o DOS dará o valor de 0050 a SP, e fará com que SS seja quatro parágrafos mais alto na memória do que o final do PSP.

29

MAIS SOBRE SEGMENTOS E ASSUME

Superposição de Segmento

Mais Sobre ASSUME

Erros de Fase

Palavras Finais

Neste capítulo final, vamos ver novamente o comando ASSUME, e como ele se relaciona com o uso dos segmentos. Juntamente com isso, aprenderemos algo sobre uma característica chamada *superposições de segmento*, de que falamos muito rapidamente. Veremos que as superposições de segmento caminham lado a lado com o comando ASSUME.

Superposição de Segmento

Até aqui, sempre lemos e escrevemos em dados localizados no segmento de dados. Como lidamos com somente um segmento neste livro (através do uso de grupos), não tínhamos motivo para ler ou escrever dados em outros segmentos.

Mas, como vimos, os programas .EXE contêm múltiplos segmentos, e até mesmo programas .COM podem conter ou usar múltiplos segmentos. Um exemplo clássico é a impressão direta na tela: muitos programas comerciais colocam dados na tela movendo-os diretamente para a memória de vídeo, "bypassando" completamente as rotinas de vídeo do ROM BIOS, por questões de velocidade. A memória de vídeo, para o IBM PC, está localizada no segmento B800h para o adaptador gráfico colorido, e no segmento B000h para o adaptador monocromático. Escrever diretamente na tela significa que queremos escrever em segmentos diferentes.

Nesta seção, vamos escrever um pequeno programa mostrando como podemos escrever em dois segmentos diferentes, usando os registradores DS e ES para apontar os dois segmentos. De fato, muitos programas que escrevem diretamente na tela usam o registrador ES como ponteiro para a memória de vídeo.

Aqui está o nosso programa. Ele é muito curto, e você pode ver que ele tem dois segmentos de dados, juntamente com uma variável em cada um desses dois:

```
SEG_DADOS       SEGMENT PUBLIC
VAR_SD          DW      1
SEG_DADOS       ENDS

SEG_EXTRA       SEGMENT PUBLIC
VAR_ES          DW      2
SEG_EXTRA       ENDS

SEG_PILHA       SEGMENT STACK
DB              10 DUP ('PILHA   ')  ;'PILHA' seguido por 3 espacos
SEG_PILHA       ENDS

SEG_COD         SEGMENT PUBLIC
TESTE           PROC    FAR
        PUSH    ES                   ;Salva endereco de retorno para RET longo
        XOR     AX,AX                ;Ha uma instrucao INT 20h em ES:0
        PUSH    AX

        MOV     AX,SEG_DADOS         ;Endereco do segmento SEG_DADOS
        MOV     DS,AX                ;Define DS para SEG_DADOS
        MOV     AX,SEG_EXTRA         ;Endereco do segmento SEG_EXTRA
        MOV     ES,AX                ;Define ES para SEG_EXTRA

        MOV     AX,VAR_SD            ;Le variavel do segmento de dados
        MOV     BX,ES:VAR_ES         ;Le variavel do segmento extra

        RET                          ;Retorna ao DOS
TESTE           ENDP

SEG_COD         ENDS

        END     TESTE
```

Usaremos esse programa para aprender sobre a pseudo-op ASSUME e superposições de segmento.

Note que colocamos os segmentos de dados e de pilha *antes* do nosso segmento de código, e também colocamos a pseudo-op ASSUME depois de todas as declarações de segmento. Como veremos nesta seção, esse arranjo é resultado direto do uso de dois segmentos de dados.

Vamos dar uma olhada nas duas instruções MOV desse programa:

```
MOV     AX,VAR_SD
MOV     BX,ES:VAR_ES
```

O ES: colocado na segunda instrução diz ao 8088 para usar o registrador ES, no lugar de DS, para essa operação (para ler o dado do nosso segmento extra). Toda instrução possui um registrador de segmento default que ela usa para referir-se a um dado. Mas, assim como fizemos com o registrador ES aqui, podemos dizer ao 8088 que queremos usar algum outro segmento para os dados.

Eis como isso funciona: o 8088 tem quatro instruções especiais, uma para cada um dos quatro registradores de segmento. Essas instruções são as instruções de superposição de segmento, e dizem ao 8088 para usar um registrador de segmento específico, ao invés do default, quando a instrução que se segue tentar ler ou escrever na memória.

Por exemplo, nossa instrução MOV AX,ES:VAR_SE está, na verdade, codificada em duas instruções. Você verá o seguinte se desassemblar o programa:

```
2CF4:0011 26          ES:
2CF4:0012 8B1E0000    MOV     BX,[0000]
```

Isso mostra que o assembler traduziu nossa instrução para uma instrução de superposição de segmento, seguida por uma instrução MOV. Agora, a instrução MOV poderá ler o dado do segmento ES, ao invés de DS.

Se você executar esse programa passo-a-passo, verá que a primeira instrução MOV coloca 1 em AX (valor de VAR_SD) e o segundo MOV coloca 2 (valor de VAR_SE) em BX. Em outras palavras: lemos dados em dois segmentos diferentes.

Mais Sobre ASSUME

Vejamos o que acontece quando você remove o ES: do programa. Altere a linha:

```
MOV     BX,ES:VAR_ES
```

de modo que fique:

```
MOV     BX,VAR_ES
```

Não estamos mais dizendo ao assembler que queremos usar o registrador ES quando lermos da memória, de modo que ele deve passar a usar o segmento default (DS), certo? Errado.

Use o Debug para ver o resultado dessa mudança. Você verá que ainda tem o segmento ES: na frente da instrução MOV. Como o assembler poderia saber que nossa variável está no segmento extra, e não no segmento de dados? Usando a informação que demos na pseudo-op ASSUME.

Nosso comando ASSUME diz ao assembler que o registrador DS aponta para o segmento SEG_DADOS, enquanto ES aponta para SEG_EXTRA. Toda vez que escrevemos uma instrução que use uma variável da memória, o assembler procura a declaração dessa variável para ver em que segmento ela está declarada. Depois, ele procura na linha da pseudo-op ASSUME qual o registrador de segmento que está apontando para esse segmento. O assembler, finalmente, usa esse registrador de segmento quando gera a instrução.

No caso da nossa instrução MOV BX,VAR_ES, o assembler verificou que VAR_ES estava no segmento chamado SEG_EXTRA, e que o registrador ES estava apontando para esse segmento, de modo que gerou uma instrução de superposição de segmento ES: por conta própria. Se tivéssemos que mover VAR_ES para o SEG_PILHA, o assembler geraria uma instrução SS:. O assembler gera automaticamente qualquer uma dessas instruções conforme a necessidade, desde que, é claro, haja um comando ASSUME para refletir o conteúdo real dos registradores de segmento.

Erros de Fase

Às vezes você encontrará uma estranha mensagem de erro gerada pelo assembler, do tipo *Phase error between passes*. Essa mensagem pode significar uma série de coisas, mas veremos um caso em particular que o ajudará a entendê-la.

Basicamente, o assembler executa uma série de passagens por um programa conforme gera sua versão em linguagem de máquina. Às vezes, como veremos aqui, o programa muda de tamanho entre uma dessas passagens.

Usando novamente nosso programa de exemplo, mova os dois segmentos de dados (SEG_DADOS e SEG_EXTRA) para que fiquem *depois* do segmento de código. O assembler agora montará a rotina principal antes de verificar os segmentos de dados. Como resultado, ele gerará uma instrução MOV normal para MOV BX,VAR_ES, pois ele não sabe que essa variável está em outro segmento.

A seguir, o assembler montará os dois segmentos de dados. Nesse ponto, ele colocará a informação de que VAR_ES está no segmento extra. Na próxima passada pelo programa, o assembler verá que agora precisa de espaço no programa para a instrução de superposição de segmento. Como ele não reservou espaço para essa instrução na primeira passagem, o assembler emite uma mensagem de erro: *Phase error between passes*.

É por isso que colocamos todos os segmentos de dados antes do segmento de código: para que o assembler soubesse quais as variáveis contidas nos segmentos. O que não está muito óbvio, no entanto, é por que colocamos o comando ASSUME em SEG_COD, e não no início desse arquivo.

Também recebemos uma mensagem de erro de fase se colocarmos o ASSUME como a primeira coisa no arquivo. Por esse motivo (que não é claro para nós), temos que declarar os segmentos *antes* da pseudo-op ASSUME, *se* formos usar qualquer indicação implícita dos segmentos. O método mais seguro, portanto, é declarar todos os dados antes do segmento de código e colocar o ASSUME dentro do segmento de código.

Palavras Finais

Até aqui você já viu muitos exemplos de programas em linguagem assembly. Durante este livro enfatizamos constantemente a programação, e não os detalhes do microprocessador 8088, dentro do seu IBM PC. Como resultado, você não conheceu todas as instruções do 8088, nem todas as pseudo-operações. Mas a maioria dos programas em assembly pode ser escrita com o que você aprendeu aqui, e nada mais. O melhor método que você pode usar para aprender mais sobre a programação em linguagem assembly é modificar os programas deste livro.

Se você imagina um outro modo de escrever qualquer parte dos nossos programas e rotinas, faça isso de qualquer jeito. Foi assim que começamos a aprender a escrever programas. Naquele tempo os programas eram em BASIC, mas a idéia continua. Encontramos programas escritos em BASIC, e começamos a aprender sobre a lingugaem em si, reescrevendo partes desses programas. Você pode fazer o mesmo com o Dskpatch.

Depois de ter experimentado com alguns exemplos, você poderá escrever seus próprios programas. Não comece do nada, por enquanto; isso é um tanto difícil para a sua primeira vez. Use os programas neste livro com uma estrutura, ou base. Não construa uma estrutura ou técnica completamente nova (o seu equivalente do projeto modular) até que se sinta bem à vontade para escrever programas em linguagem assembly.

Se você realmente se sentir atraído pela linguagem assembly, também precisará de um livro mais completo para usar como referência ao conjunto de instruções do 8088. Damos a seguir uma lista de bons livros disponíveis na época em que escrevemos este livro. Essa lista de modo algum está completa, e os livros listados aqui são somente os que nós lemos.

Os livros a seguir são boas referências para programadores:

iAPX 88 Book, Intel, 1981. Esse é o livro fonte decisivo, e uma referência muito boa.

Russel Rector e George Alexy, *The 8086 Book*, Osborne/McGraw-Hill, 1980. Esta é outra boa referência, mas é um volume grande e pesado.

Os três livros seguintes foram escritos para o IBM PC. Muita informação em cada um deles é genérica; somente os exemplos na última parte desses livros são específicos ao IBM PC. Recomendamos que você verifique todos os três em uma livraria para que escolha o mais interessante:

Scanlon, Leo J., *IBM PC & XT Assembly Language: A Guide for Programmers, Enhanced and Enlarged*, Brady Communication Co., 1985. Este livro é uma introdução completa à linguagem assembly do 8088, e é de fácil leitura. Se você ainda está um pouco vacilante em relação à linguagem assembly, esse pode ser um livro muito bom para você. Caso contrário, veja o livro do Morse.

Willen, David C., e Krantz, Jeffrey I., *8088 Assembler Language Programming: The IBM PC*, Howard W. Sam & Co., 1983. Este é um outro livro bom sobre o microprocessador 8088, escrito para o IBM PC.

Bradley, David J., *Assembly Language Programming for the IBM Personal Computer*, Prentice-Hall, 1984. O autor ajudou no projeto do IBM PC, e incluiu muitos exemplos para o IBM PC. Esses exemplos não estão completos, mas podem dar a você idéias de programas que funcionam. Ele também fala sobre assuntos mais avançados (como o processador numérico 8087) do que nos dois livros anteriores.

A próxima recomendação não é um livro de referência, e nem uma introdução ao IBM PC. É uma introdução ao microprocessador 8088, escrito por um membro do grupo de projeto da Intel:

Stephen P. Morse, *Manual do 8086/8088*, Editora Campus, 1987. Esse é um livro muito agradável. Como um dos projetistas da Intel, Morse fornece muitos detalhes do projeto do 8088 e também sobre as falhas de projeto do 8088. Embora não seja muito bom como livro de referência, essa obra é completa, além de ser muito legível e informativa.

Finalmente, o último livro é uma referência útil a qualquer um que programa o IBM PC. Pensamos nele como o compêndio de tudo que o programador pode precisar saber sobre o IBM PC e a família do microprocessador 8086.

Peter Norton, *Programmer's Guide to the IBM PC*, Microsoft Press, 1985. Inclui uma referência completa a todas as funções do DOS e BIOS, descrições de posições importantes de memória, um resumo das instruções do 8086, e diversas outras informações úteis (ou, pelo menos, interessantes).

APÊNDICE A

LISTAGEM DO DSKPATCH

Descrição das Rotinas

Listagens de Programa Para as Rotinas do Dskpatch

Arquivo Make DSKPATCH

CURSOR.ASM

ES_DISCO.ASM

CONTROLE.ASM

DISP_SET.ASM

DSKPATCH.ASM

EDITOR.ASM

ES_TEC.ASM

FANTASMA.ASM

ES_VIDEO.ASM

Este apêndice contém a versão final do Dskpatch. Se você estiver escrevendo seus próprios programas, poderá encontrar muitas rotinas de uso geral neste apêndice para ajudá-lo em seu trabalho. Incluímos descrições curtas para cada rotina.

Descrições das Rotinas

CURSOR.ASM

LIMPA_TELA — Como o comando CLS do BASIC; limpa a tela de texto.

LIMPA_ATE_FINAL_LINHA — Limpa todos os caracteres da posição do cursor até o final da linha corrente.

CURSOR_DIREITA — Move o cursor uma posição de caractere à direita, sem imprimir um espaço sobre o caractere antigo.

GOTO_XY — Muito parecido com o comando LOCATE do BASIC; move o cursor na tela.

ENVIA_CRLF — Envia o par retorno-de-carro/mudança-de-linha para a tela. Essa rotina simplesmente move o cursor para o início da linha seguinte.

ES_DISCO.ASM

PROX_SETOR — Soma um ao número de setor corrente, e depois lê esse setor para a memória e reimprime a tela.

ANT_SETOR — Lê o setor anterior. A rotina subtrai um do antigo número do setor (NR_SETOR) e lê o novo setor para a variável da memória SETOR. Ela também reimprime a tela.

LE_SETOR — Lê um setor (512 bytes) do disco para o buffer da memória, SETOR.

GRAVA_SETOR — Grava um setor (512 bytes) do buffer na memória, SETOR, para o disco.

CONTROLE.ASM

CONTROLA — O controlador central lê os caracteres do teclado e depois chama outras rotinas para fazer o trabalho real. Acrescente qualquer novo comando a TAB_CONTROLE, nesse arquivo.

DISP_SET.ASM

DISP_MEIO_SETOR — Faz o trabalho de apresentar todos os caracteres hexa e ASCII que aparecem na tela chamando DISP_LIN 16 vezes.

DISP_LIN — Apresenta uma linha da tela de meio-setor. DISP_MEIO—SETOR chama essa rotina 16 vezes para apresentar todas as 16 linhas de informação da metade do setor.

INIC_DISP_SET — Inicializa a tela do meio setor que você vê pelo Dskpatch. Esta rotina redesenha o meio setor, juntamente com os limites e os números hexa do alto, mas não imprime o cabeçalho e o pedido.

IMP_INDICES — Imprime a linha de números hexa no alto da tela do meio setor. A rotina não é muito útil para outras aplicações.

DSKPATCH.ASM

DISK_PATCH — O programa principal (bem curto) do Dskpatch. DISK_PATCH simplesmente chama uma série de outras rotinas, que fazem todo o trabalho. Também são incluídas muitas definições de variáveis usadas em todo o Dskpatch.

EDITOR.ASM

EDITA_BYTE — *Edita* um byte do meio setor alterando um byte tanto na memória (SETOR) quanto na tela. Dskpatch usa essa rotina para alterar os bytes em um setor.

ALTERA_MEMORIA — Chamada por EDITA_BYTE para alterar um único byte na memória (SETOR). Esta rotina altera o byte apontado pelo cursor fantasma.

ES_TEC.ASM

BACK_SPACE — Usado pela rotina LE_STRING para deletar um caractere, tanto da tela quanto do buffer do teclado, sempre que for pressionada a tecla Backspace.

CONVERTE_DIG_HEXA — Converte um único caractere ASCII para o seu hexadecimal equivalente. Por exemplo, a rotina converte a letra A para o número hexa 0Ah. **NOTA:** a rotina funciona somente com letras maiúsculas.

HEXA_PARA_BYTE — Converte um string de dois caracteres em um string hexa, como em A5, para um único byte com esse valor hexa. A rotina espera que os dois caracteres sejam dígitos e/ou letras maiúsculas.

LE_BYTE — Usa LE_STRING para ler um string de caracteres. Essa rotina retorna a tecla de função especial, um único caractere, ou um byte hexa se você digitou um número hexa de dois dígitos.

LE_DECIMAL — Lê um número decimal não sinalizado do teclado, usando LE_STRING para ler os caracteres. LE_DECIMAL pode ler números de 0 a 65535.

LE_STRING — Lê um string de caracteres pelo teclado ao estilo do DOS. Esta rotina também lê teclas de função especial; a função de leitura de string do DOS não faz isso.

STRING_MAIUSC — Uma rotina de uso geral, que converte um string ao estilo do DOS para letras maiúsculas.

FANTASMA.ASM

APAGA_FANTASMA — Remove os dois cursores fantasma da tela passando o atributo dos caracteres para normal (7) nas posições onde se encontra o cursor fantasma.

MOVE_POS_ASCII — Move o cursor real para o início do cursor fantasma na janela ASCII.

MOVE_POS_HEXA — Move o cursor real para o início do cursor fantasma na janela hexa.

FANTASMA_BAIXO — Move o cursor fantasma para baixo e troca para a outra metade inferior se você tentar passar da décima sexta linha.

FANTASMA_CIMA — Move o cursor fantasma para cima, ou troca a tela se você mover para acima da primeira linha da segunda metade do setor.

FANTASMA_ESQ — Move o cursor fantasma um byte à esquerda, sem ultrapassar a face esquerda de cada janela.

FANTASMA_DIR — Move o cursor fantasma um byte à direita, sem ultrapassar a face direita de cada janela.

RESTAURA_CURSOR_REAL — Move o cursor de volta a sua posição salva por SALVA_CURSOR_REAL.

SALVA_CURSOR_REAL — Salva a posição do cursor real em duas variáveis. Chame essa rotina antes de mover o cursor real se quiser recuperar sua posição quando terminarem as alterações na tela.

TELA_DESCE — Apresenta a primeira metade do setor. Você pode tentar melhorar essa versão fazendo com que a tela desça somente uma linha, com a apresentação da próxima linha superior.

TELA_SOBE — Chamado por FANTASMA_BAIXO quando você tenta mover o cursor fantasma abaixo da última linha da tela. Essa versão, na verdade, não realiza o "Scroll" da tela, mas passa para a segunda metade do cursor inteira.

IMP_FANTASMA — Desenha os cursores fantasma na tela: um na janela hexa, e um na janela ASCII. Essa rotina simplesmente altera os atributos de caracteres para 70H, para usar caracteres escuros em um fundo claro.

ES_VIDEO.ASM

Contém grande parte das rotinas de uso geral que você poderá usar nos seus próprios programas.

IMP_ATRIBUTO_N_VEZES — Uma rotina prática que você pode usar para alterar os atributos de um grupo de N caracteres. IMP_FANTASMA usa essa rotina para desenhar os cursores fantasma, e APAGA_FANTASMA a usa para remover os cursores fantasma.

IMP_CAR — Imprime um caractere na tela. Como ela usa as rotinas do ROM BIOS, essa rotina não acrescenta qualquer significado especial a quaisquer caracteres. Um caractere de retorno de carro aparecerá na tela como uma nota musical (o caractere 0Dh). Chame ENVIA_CRLF se você quiser mover o cursor para o início da linha seguinte.

IMP_CAR_N_VEZES — Imprime N cópias de um caractere na tela. Essa rotina é útil para o desenho de caracteres de linha, como usamos para as bordas da tela.

IMP_DECIMAL — Imprime uma palavra na tela como um número decimal sem sinal, na faixa de 0 a 65535.

IMP_CAB — Imprime o cabeçalho no alto da tela que você vê em Dskpatch. Lá, a rotina apresenta o número do drive de disco e o número do setor que você vê na apresentação do meio-setor.

IMP_HEXA — Apanha um número de um byte e o imprime na tela como um número hexa de dois dígitos.

IMP_DIG_HEXA — Imprime um número hexa de um dígito na tela. Essa rotina converte um nibble de quatro bits no caractere ASCII e o imprime na tela.

IMP_FORMA — Desenha quadros em volta da apresentação do meio-setor. Use IMP_FORMA para desenhar formas arbitrárias de caracteres na tela.

IMP_STRING — Uma rotina de uso geral muito útil, com a qual você pode imprimir um string de caracteres na tela. O último caractere no string deve ser um byte.

IMP_PRONTO — Imprime um string na linha de pronto, e depois limpa o restante da linha para remover quaisquer caracteres deixados anteriormente.

Listagens de Programas Para as Rotinas do Dskpatch

Arquivo Make Dskpatch

Apresentamos aqui o arquivo Make que você pode usar com o utilitário Make da Microsoft, permitindo que você tenha um sistema montado automaticamente com apenas um comando.

```
DSKPATCH.OBJ:       DSKPATCH.ASM
        MASM DSKPATCH;

ES_DISCO.OBJ:       ES_DISCO.ASM
        MASM ES_DISCO;

DISP_SET.OBJ:       DISP_SET.ASM
        MASM DISP_SET;

ES_VIDEO.OBJ:       ES_VIDEO.ASM
        MASM ES_VIDEO;

CURSOR.OBJ:         CURSOR.ASM
        MASM CURSOR;

CONTROLE.OBJ:       CONTROLE.ASM
        MASM CONTROLE;

ES_TEC.OBJ:         ES_TEC.ASM
        MASM ES_TEC;

FANTASMA.OBJ:       FANTASMA.ASM
        MASM FANTASMA;

EDITOR.OBJ:         EDITOR.ASM
        MASM EDITOR;

DSKPATCH.COM:       DSKPATCH.OBJ ES_DISCO.OBJ DISP_SET.OBJ ES_VIDEO.OBJ CURSOR.OBJ
                    CONTROLE.OBJ ES_TEC.OBJ FANTASMA.OBJ EDITOR.OBJ
        LINK @LINKINFO
        EXE2BIN DSKPATCH DSKPATCH.COM
```

CURSOR.ASM

```
CR      EQU     13                      ;Retorno de carro
LF      EQU     10                      ;Mudanca de linha

CGROUP  GROUP   CODE_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  ENVIA_CRLF

;-------------------------------------------------------------------------;
; Esta rotina envia um par CR/LF para o                                   ;
; video, usando as rotinas do DOS para que a tela seja passada           ;
; corretamente.                                                           ;
;-------------------------------------------------------------------------;
ENVIA_CRLF      PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     AH,2
        MOV     DL,CR
        INT     21h
        MOV     DL,LF
        INT     21h
        POP     DX
        POP     AX
        RET
ENVIA_CRLF      ENDP

        PUBLIC  LIMPA_TELA

;-------------------------------------------------------------------------;
; Esta rotina limpa a tela inteira.                                       ;
;-------------------------------------------------------------------------;
LIMPA_TELA      PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        XOR     AL,AL                   ;Limpa a janela inteira
        XOR     CX,CX                   ;Canto sup. esq. (0,0)
        MOV     DH,24                   ;Ultima linha = 24
        MOV     DL,79                   ;Coluna direita = 79
        MOV     BH,7                    ;Atributo normal = 7
        MOV     AH,6                    ;Chama funcao 6
        INT     10h                     ;Limpa a janela
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LIMPA_TELA      ENDP

        PUBLIC  GOTO_XY

;-------------------------------------------------------------------------;
; Esta rotina move o cursor                                               ;
;                                                                         ;
;       DH      Linha (Y)                                                 ;
;       DL      Coluna (X)                                                ;
;                                                                         ;
;-------------------------------------------------------------------------;
GOTO_XY         PROC    NEAR
        PUSH    AX
        PUSH    BX
        MOV     BH,0                    ;Pagina de video 0
        MOV     AH,2                    ;Chamada para setar posicao do cursor.
```

```
        INT     10h
        POP     BX
        POP     AX
        RET
GOTO_XY         ENDP

        PUBLIC  CURSOR_DIREITA

;-------------------------------------------------------------------------;
; Esta rotina move o cursor uma posicao a direita, ou para a              ;
; proxima linha se estiver no final de uma linha.                         ;
;                                                                         ;
; Usa:      ENVIA_CRLF                                                    ;
;-------------------------------------------------------------------------;
CURSOR_DIREITA  PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Le a posicao atual do cursor
        MOV     BH,0                    ;Pagina ativa = 0
        INT     10h                     ;Le posicao do cursor
        MOV     AH,2                    ;Define nova posicao
        INC     DL                      ;Define coluna seguinte
        CMP     DL,79                   ;Coluna deve ser <= 79
        JBE     OK
        CALL    ENVIA_CRLF              ;Vai para linha seguinte
        JMP     FIM
OK:     INT     10h
FIM:    POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
CURSOR_DIREITA  ENDP

        PUBLIC  LIMPA_ATE_FINAL_LINHA

;-------------------------------------------------------------------------;
; Esta rotina limpa a linha da posicao atual do cursor ate                ;
; o final da linha.                                                       ;
;-------------------------------------------------------------------------;
LIMPA_ATE_FINAL_LINHA   PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Le a posicao atual do cursor
        XOR     BH,BH                   ;na pagina 0
        INT     10h                     ;Agora temos (X,Y) em DL, DH
        MOV     AH,6                    ;Define limpar ate fim da linha
        XOR     AL,AL                   ;Limpa janela
        MOV     CH,DH                   ;Tudo na mesma linha
        MOV     CL,DL                   ;Comeca na posicao do cursor
        MOV     DL,79                   ;e para ao final da linha
        MOV     BH,7                    ;Usa atributo normal
        INT     10h
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LIMPA_ATE_FINAL_LINHA   ENDP

CODE_SEG        ENDS

        END
```

ES_DISCO.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  LE_SETOR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   SETOR:BYTE
        EXTRN   NR_DRIVE:BYTE
        EXTRN   NR_SETOR:WORD
DATA_SEG        ENDS

;-------------------------------------------------------------------------;
; Esta rotina le um setor (512 bytes) para SETOR.                         ;
;                                                                         ;
; Le:           NR_SETOR, NR_DRIVE                                        ;
; Escreve:      SETOR                                                     ;
;-------------------------------------------------------------------------;
LE_SETOR        PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,NR_DRIVE         ;Numero drive
        MOV     CX,1                ;Le somente 1 setor
        MOV     DX,NR_SETOR         ;Numero do setor logico
        LEA     BX,SETOR            ;Onde colocar esse setor
        INT     25h                 ;Le o setor
        POPF                        ;Descarta os flags colocados na pilha pelo DOS
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
LE_SETOR        ENDP
        PUBLIC  GRAVA_SETOR

;-------------------------------------------------------------------------;
; Esta rotina grava o setor de volta ao disco.                            ;
;                                                                         ;
; Le:           NR_DRIVE, NR_SETOR, SETOR                                 ;
;-------------------------------------------------------------------------;
GRAVA_SETOR     PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,NR_DRIVE         ;Numero drive
        MOV     CX,1                ;Grava 1 setor
        MOV     DX,NR_SETOR         ;Setor logico
        LEA     BX,SETOR
        INT     26h                 ;Grava o setor no disco
        POPF                        ;Descarta a informacao de flag
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
GRAVA_SETOR     ENDP
        PUBLIC  ANT_SETOR
        EXTRN   INIC_DISP_SET:NEAR, IMP_CAB:NEAR
        EXTRN   IMP_PRONTO:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_SETOR:WORD, EDITOR_PRONTO:BYTE
DATA_SEG        ENDS
```

```
;-------------------------------------------------------------------------;
; Esta rotina le o setor anterior, se possivel.                           ;
;                                                                         ;
; Usa:          IMP_CAB, LE_SETOR, INIC_DISP_SET                          ;
;               IMP_PRONTO                                                ;
; Le:           NR_SETOR, EDITOR_PRONTO                                   ;
; Imprime:      NR_SETOR                                                  ;
;-------------------------------------------------------------------------;
ANT_SETOR       PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     AX,NR_SETOR         ;Apanha numero do setor
        OR      AX,AX               ;Nao decrementa se for 0
        JZ      NAO_DECREMENTA
        DEC     AX
        MOV     NR_SETOR            ;Salva novo numero do setor
        CALL    IMP_CAB
        CALL    LE_SETOR
        CALL    INIC_DISP_SET       ;Apresenta novo setor
        LEA     DX,EDITOR_PRONTO
        CALL    IMP_PRONTO
NAO_DECREMENTA:
        POP     DX
        POP     AX
        RET
ANT_SETOR               ENDP

        PUBLIC  PROX_SETOR
        EXTRN   INIC_DISP_SET:NEAR, IMP_CAB:NEAR
        EXTRN   IMP_PRONTO:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_SETOR:WORD, EDITOR_PRONTO:BYTE
DATA_SEG        ENDS

;-------------------------------------------------------------------------;
; Le o proximo setor.                                                     ;
;                                                                         ;
; Usa:          IMP_CAB, LE_SETOR, INIC_DISP_SET                          ;
;               IMP_PRONTO                                                ;
; Le:           NR_SETOR, EDITOR_PRONTO                                   ;
; Imprime:      NR_SETOR                                                  ;
;-------------------------------------------------------------------------;
PROX_SETOR      PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     AX,NR_SETOR
        INC     AX                  ;Move para proximo cursor
        MOV     NR_SETOR,AX
        CALL    IMP_CAB
        CALL    LE_SETOR
        CALL    INIC_DISP_SET       ;Apresenta novo setor
        LEA     DX,EDITOR_PRONTO
        CALL    IMP_PRONTO
        POP     DX
        POP     AX
        RET
PROX_SETOR      ENDP

CODE_SEG        ENDS
        END
```

CONTROLE.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  CONTROLA
        EXTRN   LE_BYTE:NEAR, EDITA_BYTE:NEAR
        EXTRN   IMP_PRONTO:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   EDITOR_PRONTO:BYTE
DATA_SEG        ENDS

;-------------------------------------------------------------------------------
; Este e o controlador central. Durante a edicao e apresentacao normal,       ;
; esta rotina le caracteres do teclado e, se for uma tecla de                  ;
; comando (como tecla de cursor), CONTROLA chama as rotinas                    ;
; que fazem o trabalho real. Este controle e feito para as telas               ;
; especiais listadas em TAB_CONTROLE, onde os endereços das                    ;
; rotinas sao colocados apos os nomes das teclas.                              ;
;   Se o caractere nao for especial, entao deve ser colocado                   ;
; diretamente no buffer do setor -- este e o modo de edicao.                   ;
;                                                                              ;
; Usa:    LE_BYTE, EDITA_BYTE, IMP_PRONTO                                      ;
; Le:     EDITOR_PRONTO                                                        ;
;-------------------------------------------------------------------------------
CONTROLA        PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    DX
LOOP_CONT:
        CALL    LE_BYTE                 ;Le caractere em AL
        OR      AH,AH                   ;AH = 0 se nenhum caractere lido
                                        ;   -1 se codigo estendido
        JZ      NENHUM_CAR_LIDO         ;Nenhum caractere, tenta de novo
        JS      TECLA_ESPECIAL          ;Le codigo estendido
        MOV     DL,AL
        CALL    EDITA_BYTE              ;Foi caractere normal, edita byte.
        JMP     LOOP_CONT               ;Le outro caractere

TECLA_ESPECIAL:
        CMP     AL,68                   ;F10--saida?
        JE      FIM_CONTROLE            ;Sim, sai
                                        ;Usa BX para verificar tabela
        LEA     BX,TAB_CONTROLE
LOOP_ESPECIAL:
        CMP     BYTE PTR [BX],0         ;Fim da tabela?
        JE      FORA_TABELA             ;Sim, tecla nao estava na tabela
        CMP     AL,[BX]                 ;Achou o codigo entrado?
        JE      CONTROLE                ;Sim, envia
        ADD     BX,3                    ;Nao, tenta proxima entrada
        JMP     LOOP_ESPECIAL           ;Verifica proxima entrada na tabela

CONTROLE:
        INC     BX                      ;Aponta endereco na rotina
        CALL    WORT PTR [BX]           ;Chama rotina
        JMP     LOOP_CONT               ;Espera outra tecla

FORA_TABELA:                            ;Somente le o proximo caractere
        JMP     LOOP_CONT

NENHUM_CAR_LIDO:
        LEA     DX,EDITOR_PRONTO
        CALL    IMP_PRONTO              ;Apaga caracteres invalidos digitados
        JMP     LOOP_CONT               ;Tenta de novo
FIM_CONTROLE:
        POP     DX
```

```
        POP     BX
        POP     AX
        RET
CONTROLA        ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
DATA_SEG        SEGMENT PUBLIC
        EXTRN   LE_SETOR:NEAR                                   ;Em ES_DISCO.ASM
        EXTRN   ANT_SETOR:NEAR                                  ;Em ES_DISCO.ASM
        EXTRN   FANTASMA_CIMA:NEAR, FANTASMA_BAIXO:NEAR         ;Em FANTASMA.ASM
        EXTRN   FANTASMA_ESQ:NEAR, FANTASMA_DIR:NEAR
        EXTRN   GRAVA_SETOR:NEAR                                ;Em ES_DISCO.ASM

;-------------------------------------------------------------------------
; Esta tabela contem as teclas ASCII estendidas e os enderecos           ;
; das rotinas que devem ser chamadas quadno as teclas forem pressionadas ;
;       O formato da tabela e:                                           ;
;           DB      60          Codigo estendido para F2                 ;
;           DW      OFFSET CGROUP:PROX_SETOR                             ;
;-------------------------------------------------------------------------
TAB_CONTROLE    LABEL   BYTE
        DB      59                                              ;F1
        DW      OFFSET CGROUP:ANT_SETOR
        DB      60                                              ;F2
        DW      OFFSET CGROUP:PROX_SETOR
        DB      72                                              ;Cursor acima
        DW      OFFSET CGROUP:FANTASMA_CIMA
        DB      80                                              ;Cursor abaixo
        DW      OFFSET CGROUP:FANTASMA_BAIXO
        DB      75                                              ;Cursor esquerda
        DW      OFFSET CGROUP:FANTASMA_ESQ
        DB      77                                              ;Cursor direita
        DW      OFFSET CGROUP:FANTASMA_DIR
        DB      88                                              ;Shift F5
        DW      OFFSET CGROUP:GRAVA_SETOR
        DB      0                                               ;Fim da tabela
DATA_SEG        ENDS
        END
```

DISP_SET.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG   ;Agrupa dois segmentos
        ASSUME  CS:CGROUP, DS:CGROUP

;-------------------------------------------------------------------------;
; Caracteres graficos para o quadro do setor.                             ;
;-------------------------------------------------------------------------;
BARRA_VERT      EQU     0BAh
BARRA_HOR       EQU     0CDh
SUP_ESQ         EQU     0C9h
SUP_DIR         EQU     0BBh
INF_ESQ         EQU     0C8h
INF_DIR         EQU     0BCh
T_ALTO          EQU     0CBh
T_BAIXO         EQU     0CAh
TRACO_ALTO      EQU     0D1h
TRACO_BAIXO     EQU     0CFh

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  INIC_DISP_SET
        EXTRN   IMP_FORMA:NEAR, ENVIA_CRLF:NEAR
        EXTRN   GOTO_XY:NEAR, IMP_FANTASMA:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   DESL_SETOR:WORD
DATA_SEG        ENDS

;-------------------------------------------------------------------------;
; Esta rotina inicializa a apresentacao do setor.                         ;
;                                                                         ;
; Usa:          IMP_FORMA, ENVIA_CRLF, DISP_MEIO_SETOR                    ;
;               IMP_INDICES, GOTO_XY, IMP_FANTASMA                        ;
; Le:           FORMA_LIN_SUP, FORMA_LIN_INF                              ;
; Escreve:      DESL_SETOR                                                ;
;-------------------------------------------------------------------------;
INIC_DISP_SET   PROC    NEAR
        PUSH    DX
        XOR     DL,DL                   ;Move cursor para posicao inicial
        MOV     DH,LINHAS_ANTES_SETOR
        CALL    GOTO_XY
        CALL    IMP_INDICES
        LEA     DX,FORMA_LIN_SUP
        CALL    IMP_FORMA
        CALL    ENVIA_CRLF
        XOR     DX,DX                   ;Começa no inicio do setor
        MOV     DESL_SETOR              ;Deslocamento do setor = 0
        CALL    DISP_MEIO_SETOR
        LEA     FORMA_LIN_INF
        CALL    IMP_FORMA
        CALL    IMP_FANTASMA            ;Imprime o cursor fantasma
        POP     DX
        RET
INIC_DISP_SET   ENDP

        EXTRN   IMP_CAR_N_VEZES:NEAR, IMP_HEXA:NEAR, IMP_CAR:NEAR
        EXTRN   IMP_DIG_HEXA:NEAR, ENVIA_CRLF

;-------------------------------------------------------------------------;
; Esta rotina escreve os indices (0 a F) no topo da apresentação          ;
; do meio setor.                                                          ;
;                                                                         ;
; Usa:          IMP_CAR_N_VEZES, IMP_HEXA, IMP_CAR                        ;
;               IMP_DIG_HEXA, ENVIA_CRLF                                  ;
;-------------------------------------------------------------------------;
IMP_INDICES     PROC    NEAR
        PUSH    CX
        PUSH    D
```

```
        MOV     DL,' '                  ;Imprime 9 espacos para o lado esquerdo
        MOV     CX,9
        CALL    IMP_CAR_N_VEZES
        XOR     DH,DH                   ;Inicia com 0
LOOP_NR_HEXA:
        MOV     DL,DH
        CALL    IMP_HEXA
        MOV     DL,' '
        CALL    IMP_CAR
        INC     DH
        CMP     DH,10h                  ;Terminou?
        JB      LOOP_NR_HEXA

        MOV     DL,' '                  ;Imprime os digitos sobre a janela ASCII
        MOV     CX,2
        CALL    IMP_CAR_N_VEZES
        XOR     DL,DL
LOOP_DIG_HEXA:
        CALL    IMP_DIG_HEXA
        INC     DL
        CMP     DL,10h
        JB      LOOP_DIG_HEXA
        CALL    ENVIA_CRLF
        POP     DX
        POP     CX
        RET
IMP_INDICES     ENDP

        PUBLIC  DISP_MEIO_SETOR
        EXTRN   ENVIA_CRLF:NEAR

;-------------------------------------------------------------------------
; Esta rotina apresenta meio setor (256 bytes)                           ;
;                                                                        ;
;       DS:DX   Deslocamento no setor, em bytes -- deve se multiplo de 16 ;
;                                                                        ;
; Usa:          DISP_LIN, ENVIA_CRLF                                     ;
;-------------------------------------------------------------------------
DISP_MEIO_SETOR         PROC    NEAR
        PUSH    CX
        PUSH    DX
        MOV     CX,16                   ;Mostra 16 linhas
MEIO_SETOR:
        CALL    DISP_LIN
        CALL    ENVIA_CRLF
        ADD     DX,16
        LOOP    MEIO_SETOR
        POP     DX
        POP     CX
        RET
DISP_MEIO_SETOR         ENDP
        PUBLIC  DISP_LIN
        EXTRN   IMP_HEXA:NEAR
        EXTRN   IMP_CAR:NEAR
        EXTRN   IMP_CAR_N_VEZES:NEAR

;-------------------------------------------------------------------------
; Esta rotina mostra uma linha de dados, ou 16 bytes, primeiro em hexa,  ;
; e depois em ASCII.                                                     ;
;                                                                        ;
;       DS:DX   Deslocamento no setor, em bytes                          ;
;                                                                        ;
; Usa:          IMP_CAR, IMP_HEXA, IMP_CAR_N_VEZES                       ;
; Le:             SETOR                                                  ;
;-------------------------------------------------------------------------
DISP_LIN        PROC    NEAR
        PUSH    BX
        PUSH    CX
```

```
        PUSH    DX
        MOV     BX,DX                   ;Deslocamento mais util em BX
        MOV     DL,' '
        MOV     CX,3                    ;Imprime 3 espacos antes da linha
        CALL    IMP_CAR_N_VEZES
                                        ;Imprime deslocamento em hexa
        CMP     BX,100h                 ;Primeiro digito = 1?
        JB      IMP_UM                  ;Nao, espaco em branco ja em DL
        MOV     DL,'1'                  ;Sim, coloca "1" em DL para saida hexa
IMP_UM:
        CALL    IMP_CAR
        MOV     DL,BL                   ;Copia byte baixo em DL para saida hexa
        CALL    IMP_HEXA
                                        ;Separa a escrita
        MOV     DL,' '
        CALL    IMP_CAR
        MOV     BARRA_VERT              ;Desenha lado esquerdo do quadro
        CALL    IMP_CAR
        MOV     DL,' '
        CALL    IMP_CAR
                                        ;Agora imprime 16 bytes
        MOV     CX,16                   ;Copia os 16 bytes
        PUSH    BX                      ;Salva o deslocamento para LOOP_ASCII
LOOP_HEXA:
        MOV     DL,SETOR                ;Pega 1 byte
        CALL    IMP_HEXA                ;Copia o byte em hexa
        MOV     DL,' '                  ;Espaco entre os numeros
        CALL    IMP_CAR
        INC     BX
        LOOP    LOOP_HEXA

        MOV     DL,BARRA_VERT           ;Separa a escrita
        CALL    IMP_CAR
        MOV     DL,' '                  ;Acrescenta outro espaco antes dos caracteres
        CALL    IMP_CAR
        MOV     CX,16
        POP     BX                      ;Volta com o deslocamento em SETOR
LOOP_ASCII:
        MOV     DL,SETOR[BX]
        CALL    IMP_CAR
        INC     BX
        LOOP    LOOP_ASCII

        MOV     DL,' '                  ;Desenha lado direito do quadro
        CALL    IMP_CAR
        MOV     DL,BARRA_VERT
        CALL    IMP_CAR

        POP     DX
        POP     CX
        POP     BX
        RET
DISP_LIN        ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
        EXTRN   SETOR:BYTE

FORMA_LIN_SUP           LABEL   BYTE
        DB      ' ',7
        DB      SUP_ESQ,1
        DB      BARRA_HOR,12
        DB      TRACO_ALTO,1
        DB      BARRA_HOR,11
        DB      TRACO_ALTO,1
        DB      BARRA_HOR,11
        DB      TRACO_ALTO,1
```

```
        DB      BARRA_HOR,12
        DB      T_ALTO,1
        DB      BARRA_HOR,18
        DB      SUP_DIR,1
        DB      0
FORMA_LIN_INF       LABEL   BYTE
        DB      ' ',7
        DB      INF_ESQ,1
        DB      BARRA_HOR,12
        DB      TRACO_BAIXO,1
        DB      BARRA_HOR,11
        DB      TRACO_BAIXO,1
        DB      BARRA_HOR,11
        DB      TRACO_BAIXO,1
        DB      BARRA_HOR,12
        DB      T_BAIXO,1
        DB      BARRA_HOR,18
        DB      INF_DIR
        DB      0
DATA_SEG        ENDS

        END
```

DSKPATCH.ASM

```
CGROUP    GROUP    CODE_SEG, DATA_SEG
          ASSUME   CS:CGROUP, DS:CGROUP

CODE_SEG           SEGMENT PUBLIC
          ORG      100h

          EXTRN    LIMPA_TELA:NEAR, LE_SETOR:NEAR
          EXTRN    INIC_DISP_SET:NEAR, IMP_CAB:NEAR
          EXTRN    IMP_PRONTO:NEAR, CONTROLA:NEAR
DISK_PATCH         PROC     NEAR
          CALL     LIMPA_TELA
          CALL     IMP_CAB
          CALL     LE_SETOR
          CALL     INIC_DISP_SET
          LEA      DX,EDITOR_PRONTO
          CALL     IMP_PRONTO
          CALL     CONTROLA
          INT      20h
DISK_PATCH         ENDP

CODE_SEG           ENDS

DATA_SEG           SEGMENT PUBLIC

          PUBLIC   DESL_SETOR

;-------------------------------------------------------------------------;
; DESL_SETOR e o deslocamento ao meio                                     ;
; setor dentro do setor completo. Ele deve ser                            ;
; multiplo de 16 e nao maior que 256                                      ;
;-------------------------------------------------------------------------;
DESL_SETOR         DW       0

          PUBLIC   NR_SETOR, NR_DRIVE
NR_SETOR           DW   0              ;Inicialmente setor 0
NR_DRIVE           DB   0              ;Inicialmente Drive A:

          PUBLIC   LINHAS_ANTES_SETOR, NR_LINHA_CAB
          PUBLIC   CAB_1, CAB_2

;-------------------------------------------------------------------------;
; LINHAS_ANTES_SETOR e o numero de linhas                                 ;
; no alto da tela antes da apresentacao do                                ;
; meio setor.                                                             ;
;-------------------------------------------------------------------------;
LINHAS_ANTES_SETOR      DB       2
NR_LINHA_CAB       DB       0
CAB_1              DB       'Disco ',0
CAB_2              DB                Setor ',0
          PUBLIC   NR_LINHA_PRONTO, EDITOR_PRONTO
NR_LINHA_PRONTO         DB       21
EDITOR_PRONTO      DB       'Press tecla funcao, ou entre '
                            ' caractere ou byte hexa: ',0

          PUBLIC   SETOR

;-------------------------------------------------------------------------;
; O setor inteiro (ate 8192 bytes) esta                                   ;
; armazenado nessa parte da memoria.                                      ;
;-------------------------------------------------------------------------;
SETOR     DB       8192 DUP (0)

DATA_SEG           ENDS

          END      DISK_PATCH
```

EDITOR.ASM

```
CGROUP    GROUP     CODE_SEG, DATA_SEG
          ASSUME    CS:CGROUP, DS:CGROUP

CODE_SEG            SEGMENT   PUBLIC

DATA_SEG            SEGMENT   PUBLIC
          EXTRN     SETOR:BYTE
          EXTRN     DESL_SETOR:WORD
          EXTRN     CURSOR_FANT_X:BYTE
          EXTRN     CURSOR_FANT_Y:BYTE
DATA_SEG            ENDS

;-------------------------------------------------------------------------;
; Esta rotina coloca um byte em SETOR, para posicao de memoria            ;
; apontado pelo cursor fantasma.                                          ;
;                                                                         ;
;     DL      Byte a ser alocado em SETOR                                 ;
;                                                                         ;
; O deslocamento e calculado por                                          ;
;   DESL = DESL_SETOR + (17 * CURSOR_FANT_Y) + CURSOR_FANT_X              ;
;                                                                         ;
; Le:     CURSOR_FANT_X, CURSOR_FANT_Y, DESL_SETOR                        ;
; Altera: SETOR                                                           ;
;-------------------------------------------------------------------------;
ALTERA_MEMORIA  PROC      NEAR
          PUSH      AX
          PUSH      BX
          PUSH      CX
          MOV       BX,DESL_SETOR
          MOV       AL,CURSOR_FANT_Y
          XOR       AH,AH
          MOV       CL,4                    ;Multiplica CURSOR_FANT_Y por 16
          SHL       AX,CL
          ADD       BX,AX                   ;BX = DESL_SETOR + (16 * Y)
          MOV       AL,CURSOR_FANT_X
          XOR       AH,AH
          ADD       BX,AX                   ;Eis o endereco!
          MOV       SETOR[BX],DL            ;Agora, armazena o byte.
          POP       CX
          POP       BX
          POP       AX
          RET
ALTERA_MEMORIA          ENDP

          PUBLIC    EDITA_BYTE
          EXTRN     SALVA_CURSOR_REAL:NEAR, RESTAURA_CURSOR_REAL:NEAR
          EXTRN     MOVE_POS_HEXA:NEAR, MOVE_POS_ASCII:NEAR
          EXTRN     IMP_FANTASMA:NEAR, IMP_PRONTO:NEAR
          EXTRN     CURSOR_DIREITA:NEAR, IMP_HEXA:NEAR, IMP_CAR:NEAR
DATA_SEG            SEGMENT PUBLIC
          EXTRN     EDITOR_PRONTO:BYTE
DATA_SEG            ENDS

;-------------------------------------------------------------------------;
; Esta rotina troca o byte na memoria e na tela                           ;
;                                                                         ;
;     DL      Byte a colocar em SETOR, e alterar na tela                  ;
;                                                                         ;
; Usa:        SALVA_CURSOR_REAL, RESTAURA_CURSOR_REAL                     ;
;             MOVE_POS_HEXA, MOVE_POS_ASCII                               ;
;             IMP_FANTASMA, IMP_PRONTO, CURSOR_DIREITA                    ;
;             IMP_HEXA, IMP_CAR, ALTERA_MEMORIA                           ;
; Le:         EDITOR_PRONTO                                               ;
;-------------------------------------------------------------------------;
EDITA_BYTE          PROC      NEAR
          PUSH      DX
```

```
        CALL    SALVA_CURSOR_REAL
        CALL    MOV_POS_HEXA            ;Move para o numero hexa
        CALL    CURSOR_DIREITA          ;  na janela hexa
        CALL    IMP_HEXA                ;Imprime o novo numero
        CALL    MOVE_POS_ASCII          ;Move o caractere na janela ASCII
        CALL    IMP_CAR                 ;Imprime o novo caractere
        CALL    RESTAURA_CURSOR_REAL    ;Move o cursor de volta para onde estava
        CALL    IMP_FANTASMA            ;Reimprime o cursor fantasma
        CALL    ALTERA_MEMORIA          ;Salva este novo byte em SETOR
        LEA     DX,EDITOR_PRONTO
        CALL    IMP_PRONTO
        POP     DX
        RET
EDITA_BYTE      ENDP

CODE_SEG        ENDS

        END
```

ES_TEC.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

BS      EQU     8                       ;Caractere de backspace
CR      EQU     13                      ;Caractere retorno_de_carro
ESC     EQU     27                      ;Caractere Escape

CODE_SEG        SEGMENT PUBLIC

        PUBLIC  STRING_MAIUSC

;-------------------------------------------------------------------------;
; Esta rotina converte o string, usando o formato do DOS para strings,    ;
; em letras maiúsculas.                                                   ;
;                                                                         ;
;       DS:DX     Endereço do buffer do string                            ;
;-------------------------------------------------------------------------;
STRING_MAIUSC       PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX                      ;Contador de caracteres
        MOV     CL,[BX]                 ;no segundo byte do buffer
        XOR     CH,CH                   ;Zera byte alto do contador
LOOP_MAIUSC:
        INC     BX                      ;Aponta prox. caractere
        MOV     AL,[BX]
        CMP     AL,'a'                  ;Ve se e letra minuscula
        JB      NAO_MINUSC              ;Nao
        CMP     AL,'z'
        JA      NAO_MINUSC
        ADD     AL,'A'-'a'              ;Converte para maiuscula
        MOV     [BX],AL
NAO_MINUSC:
        LOOP    LOOP_MAIUSC
        POP     CX
        POP     BX
        POP     AX
        RET
STRING_MAIUSC           ENDP

;-------------------------------------------------------------------------;
; Esta rotina converte um caractere de ASCII (hexa) para um nibble        ;
; (4 bits)                                                                ;
;                                                                         ;
;           AL      Caractere a converter                                 ;
; Retorna:  AL      Nibble                                                ;
;           CF      Setado se houver erro                                 ;
;-------------------------------------------------------------------------;
CONVERTE_DIG_HEXA       PROC    NEAR
        CMP     AL,'0'                  ;Digito legal?
        JB      ERRO_DIG                ;Nao
        CMP     AL,'9'                  ;Nao esta certo ainda
        JA      TENTA_HEXA              ;Poderia ser digito hexa
        SUB     AL,'0'                  ;Se digito decimal, converte para nibble
        CLC                             ;Limpa o carry, nenhum erro
        RET
TENTA_HEXA:
        CMP     AL,'A'                  ;Nao tem certeza
        JB      ERRO_DIG                ;Nao hexa
        CMP     AL,'F'                  ;Nao tem certeza
        JA      ERRO_DIG                ;Nao hexa
        SUB     AL,'A'-10               ;Hexa, converte para nibble
        CLC                             ;Limpa o carry
        RET
```

```
ERRO_DIG:
        STC                             ;Seta o carry, erro
        RET
CONVERTE_DIG_HEXA       ENDP

        PUBLIC  HEXA_PARA_BYTE

;-------------------------------------------------------------------------
; Esta rotina converte os dois caracteres em DS:DX de hexa para
; um byte.
;
;       DS:DX       Endereco dos dois caracteres do numero hexa
; Retorna:
;       AL          Byte
;       CF          Setado se houver erro
;
; Usa:              CONVERTE_DIG_HEXA
;-------------------------------------------------------------------------
HEXA_PARA_BYTEPROC      NEAR
        PUSH    BX
        PUSH    CX
        MOV     BX,DX                   ;Coloca endereco em BX para endereco indireto
        MOV     AL,[BX]                 ;Apanha primeiro digito
        CALL    CONVERTE_DIG_HEXA
        JC      ERRO_HEXA               ;Digito hexa errado se flag setado
        MOV     CX,4                    ;Agora multiplica por 16
        SHL     AL,CL
        MOV     AH,AL                   ;Retem uma copia
        INC     BX                      ;Apanha segundo digito
        MOV     AL,[BX]
        CALL    CONVERTE_DIG_HEXA
        JC      ERRO_HEXA               ;Digito hexa errado se carry setado
        OR      AL,AH                   ;Combina dois nibbles
        CLC                             ;Limpa carry se nao houver erro
FIM_HEXA:
        POP     CX
        POP     BX
        RET
ERRO_HEXA:
        STC                             ;Seta carry, indicando erro
        JMP     FIM_HEXA
HEXA_PARA_BYTE  PROC    NEAR
        PUBLIC  LE_STRING
        EXTRN   IMP_CAR:NEAR

;-------------------------------------------------------------------------
; Esta rotina e semelhante a funcao 0Ah do DOS. Mas essa
; funcao retorna um caractere especial se uma tecla de
; funcao for pressionada -- sem retornar essas teclas.
; E ESC apagara a entrada e comecara de novo.
;
;       DS:DX   Endereço do buffer do teclado. O primeiro byte deve
;               conter um maximo de caracteres a ler (mais
;               o CR). E o segundo byte sera usado por esta
;               rotina para retornar o numero de caracteres
;               realmente lidos.
;
;                       0       Nenhum caractere lido
;                       -1      Um caractere lido de
;                       outra forma, numero realmente lido (sem a
;                               tecla Enter)
; Usa:          BACK_SPACE, IMP_CAR
;-------------------------------------------------------------------------

LE_STRING       PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    SI
```

```
        MOV     SI,DX                   ;Usa SI para reg. de indice e
INICIO:
        MOV     BX,2                    ;BX para deslocamento no buffer
        MOV     AH,7                    ;Chama para entrada sem checar
        INT     21h                     ;CTRL_BREAK e sem eco
        OR      AL,AL                   ;E um caractere estendido?
        JZ      ESTENDIDO               ;Sim, le o caractere estendido
NAO_ESTENDIDO:                          ;Caractere estendido e erro se buffer nao vazio
        CMP     AL,CR                   ;E um retorno de carro (CR)?
        JE      FIM_ENTRADA             ;Sim, fim da entrada
        CMP     AL,BS                   ;E um retrocesso (BS)?
        JNE     NAO_BS                  ;Nao
        CALL    BACK_SPACE              ;Sim, deleta caractere
        CMP     BL,2                    ;Buffer esta vazio?
        JE      INICIO                  ;Sim, pode ler caracteres estendidos de novo
        JMP     SHORT LE_PROX_CAR       ;Nao, continua lendo normalmente
NAO_BS: CMP     AL,ESC                  ;E um ESC -- apagar buffer?
        JE      APAGA_BUFFER            ;Sim, apaga o buffer
        CMP     BL,[SI]                 ;Verifica se buffer esta cheio
        JA      BUFFER_CHEIO            ;Buffer cheio
        MOV     [SI+BX],AL              ;Senao, salva caractere no buffer
        INC     BX                      ;Aponta prox. byte livre no buffer
        PUSH    DX
        MOV     DL,AL                   ;Ecoa caractere na tela
        CALL    IMP_CAR
        POP     DX
LE_PROX_CAR:
        MOV     AH,7
        INT     21h
        OR      AL,AL                   ;Um caractere estendido nao e valido
                                        ;   quando o buffer nao esta vazio
        JNE     NAO_ESTENDIDO           ;Caractere valido
        MOV     AH,7
        INT     21h                     ;Elimina caractere estendido

;-----------------------------------------------------------------------
; Indica uma condicao de erro por meio de                              ;
; um beep no console: chr$(7).                                         ;
;-----------------------------------------------------------------------
INDICA_ERRO:
        PUSH    DX
        MOV     DL,7                    ;Avisa com beep (chr$(7))
        MOV     AH,2
        INT     21h
        POP     DX
        JMP     SHORT LE_PROX_CAR       ;Agora le o proximo caractere

;-----------------------------------------------------------------------
; Esvazia o buffer de string e apaga todos                             ;
; os caracteres apresentados na tela.                                  ;
;-----------------------------------------------------------------------
APAGA_BUFFER:
        PUSH    CX
        MOV     CL,[SI]                 ;Retrocede numero maximo de
        XOR     CH,CH
LOOP_APAGA:                             ;   caracteres no buffer. BACK_SPACE
        CALL    BACK_SPACE              ;evitara que o cursor se
        LOOP    LOOP_APAGA              ;   mova muito para tras
        MOV     CX
        JMP     INICIO                  ;Agora pode ler caracteres ASCII estendidos,
                                        ;pois o buffer esta vazio

;-----------------------------------------------------------------------
; O buffer estava cheio, e nao podemos ler                             ;
; outro caractere. Envia um beep para alertar                          ;
; o usuario da condicao do buffer.                                     ;
;-----------------------------------------------------------------------
```

```
BUFFER_CHEIO:
        JMP     SHORT INDICA_ERRO       ;Se buffer cheio, sinaliza

;-------------------------------------------------------------------------;
; Le o codigo estendido e o coloca no buffer                              ;
; como unico caractere, e depois retorna                                  ;
; -1 como numero de caracteres lidos.                                     ;
;-------------------------------------------------------------------------;
ESTENDIDO:                              ;Le um codigo ASCII estendido
        MOV     AH,7
        INT     21h
        MOV     [SI+2],AL               ;Coloca esse caractere no buffer
        MOV     BL,0FFh                 ;Num. caract. lidos = -1 para especiais
        JMP     SHORT FIM_STRING

;-------------------------------------------------------------------------;
; Salva a contagem do numero de caracteres                                ;
; lidos e retorna.                                                        ;
;-------------------------------------------------------------------------;
FIM_ENTRADA:                            ;Termina entrada
        SUB     BL,2                    ;Contador de caracteres lidos
FIM_STRING:
        MOV     [SI+1],BL               ;Retorna numero de caracteres
        POP     SI
        POP     BX
        POP     AX
        RET
LE_STRING       ENDP

        PUBLIC  LE_BYTE

;-------------------------------------------------------------------------;
; Esta rotina le um unico caractere ASCII ou um numero hexa de 2 digitos. ;
;                                                                         ;
; Retorna byte em       AL      Codigo do caractere (a menos que AH = 0)  ;
;                       AH      1 se leu caract. ASCII ou numero hexa     ;
;                               0 se nenhum caractere                     ;
;                               -1 se leu tecla especial                  ;
;                                                                         ;
; Usa:          HEXA_PARA_BYTE, STRING_MAIUSC, LE_STRING                  ;
; Le:           ENTRA_TECLADO, etc.                                       ;
; Escreve:      ENTRA_TECLADO, etc.                                       ;
;-------------------------------------------------------------------------;
LE_BYTE         PROC    NEAR
        PUSH    DX
        MOV     LIMITE_NUM_CARACT,3             ;Permite somente 2 caracteres (mais Enter)
        LEA     DX,ENTRA_TECLADO
        CALL    LE_STRING
        CMP     NUM_CARACT_LIDOS,1              ;Ve quantos caracteres
        JE      ENTRADA_ASCII                   ;So um, tratar como caractere ASCII
        JB      SEM_CARACTERES                  ;So ENTER pressionado
        CMP     BYTE PTR NUM_CARACT_LIDOS,0FFh  ;Tecla especial?
        JE      TECLA_ESPECIAL                  ;Sim
        CALL    STRING_MAIUSC                   ;Nao, converte para maiusculas
        LEA     DX,STRING                       ;Endereco do string a converter
        CALL    HEXA_PARA_BYTE                  ;Converte string de hexa para byte
        JC      SEM_CARACTERES                  ;Erro, retorna nenhum caractere lido
        MOV     AH,1                            ;Indica que leu um caractere
FIM_LEITURA:
        POP     DX
        RET
SEM_CARACTERES:
        XOR     AH,AH                           ;Indica 'nenhum caractere lido'
        JMP     FIM_LEITURA
ENTRADA_ASCII:
        MOV     AL,STRING                       ;Carrega caractere lido
        MOV     AH,1                            ;Indica que leu 1 caractere
        JMP     FIM_LEITURA
```

```
TECLA_ESPECIAL:
        MOV     AL,STRING[0]                    ;Retorna codigo de varredura
        MOV     AH,0FFh                         ;Indica tecla especial com -1
        JMP     FIM_LEITURA
LE_BYTE         ENDP

        PUBLIC  LE_DECIMAL

;-------------------------------------------------------------------------;
; Esta rotina apanha o buffer de saida de LE_STRING e converte            ;
; o string de digitos decimais para uma palavra.                          ;
;                                                                         ;
;       AX      Palavra convertida de decimal                             ;
;       CF      Setado se houver erro                                     ;
;                                                                         ;
; Usa:          LE_STRING                                                 ;
; Le:           ENTRA_TECLADO, etc.                                       ;
; Escreve:      ENTRA_TECLADO, etc.                                       ;
;-------------------------------------------------------------------------;
LE_DECIMAL      PROC    NEAR
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     LIMITE_NUM_CARACT,6     ;Numero maximo 5 digitos (65535)
        LEA     DX,ENTRA_TECLADO
        CALL    LE_STRING
        MOV     CL,NUM_CARACT_LIDOS     ;Apanha numero de caracteres lidos
        XOR     CH,CH                   ;Zera byte alto do contador
        CMP     CL,0                    ;Retorna erro se nenhum caractere lido
        JLE     ERRO_DIG_DECIMAL
        XOR     AX,AX                   ;Comeca com numero zero
        XOR     BX,BX                   ;e no inicio do string
CONVERTE_DIGITO:
        MOV     DX,10                   ;Multiplica numero por 10
        MUL     DX                      ;Multiplica AX por 10
        JC      ERRO_DIG_DECIMAL        ;CF setado se MUL estourou uma palavra
        MOV     DL,STRING[BX]           ;Apanha proximo digito
        SUB     DL,'0'                  ;E converte para um nibble (4 bits)
        JS      ERRO_DIG_DECIMAL        ;Digito errado se < 0
        CMP     DL,9                    ;E um digito errado?
        JA      ERRO_DIG_DECIMAL        ;Sim
        ADD     AX,DX                   ;Nao, entao soma ao numero
        INC     BX                      ;Aponta proximo caractere
        LOOP    CONVERTE_DIGITO         ;Apanha proximo digito
FIM_DECIMAL:
        POP     DX
        POP     CX
        POP     BX
        RET
ERRO_DIG_DECIMAL:
        STC                             ;Seta carry para avisar erro
        JMP     FIM_DECIMAL
LE_DECIMAL      ENDP

        PUBLIC  BACK_SPACE
        EXTRN   IMP_CAR:NEAR

;-------------------------------------------------------------------------;
; Esta rotina deleta um caractere de cada vez, do buffer e da tela        ;
; quando o buffer nao esta vazio. BACK_SPACE simplesmente                 ;
; retorna quando o buffer esta vazio.                                     ;
;                                                                         ;
;       DS:SI+BX        Caractere mais recente do buffer                  ;
;                                                                         ;
; Usa:          IMP_CAR                                                   ;
;-------------------------------------------------------------------------;
```

```
BACK_SPACE      PROC    NEAR        ;Delete um caractere
        PUSH    AX
        PUSH    DX
        CMP     BX,2                ;Buffer vazio?
        JE      FIM_BS              ;Sim, le o prox. caractere
        DEC     BX                  ;Remove um caractere do buffer
        MOV     AH,2                ;Remove caractere da tela
        MOV     DL,BS
        INT     21h
        MOV     DL,20h              ;Imprime espaco aqui
        CALL    IMP_CAR
        MOV     DL,BS               ;Retrocede
        INT     21h
FIM_BS: POP     DX
        POP     AX
        RET
BACK_SPACE      ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
ENTRA_TECLADO   LABEL   BYTE
LIMITE_NUM_CARACT   DB      0       ;Tamanho do buffer de entrada
NUM_CARACT_LIDOS    DB      0       ;Numero de caracteres lidos
STRING          DB      80 DUP (0)  ;Um buffer para entrada pelo teclado
DATA_SEG        ENDS

        END
```

FANTASMA.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        PUBLIC  GOTO_XY:NEAR

DATA_SEG        SEGMENT PUBLIC
        EXTRN   LINHAS_ANTES_SETOR
DATA_SEG        ENDS

;-------------------------------------------------------------------------;
; Esta rotina move o cursor real para a posicao do cursor                 ;
; fantasma na janela hexa.                                                ;
;                                                                         ;
; Usa:      GOTO_XY                                                       ;
; Le:       LINHAS_ANTES_SETOR, CURSOR_FANT_X, CURSOR_FANT_Y              ;
;-------------------------------------------------------------------------;

MOVE_POS_HEXA PROC      NEAR
        PUSH    AX
        PUSH    CX
        PUSH    DX
        MOV     DH,LINHAS_ANTES_SETOR   ;Encontra linha do fantasma (0,0)
        ADD     DH,2                    ;Mais linha de hexa e barra horizontal
        ADD     DH,CURSOR_FANT_Y        ;DH = linha do cursor
        MOV     DL,8                    ;Indenta a esquerda
        MOV     CL,3                    ;Cada coluna usa 3 caracteres,
        MOV     AL,CURSOR_FANT_X        ;e devemos multiplicar CURSOR_X por 3
        MUL     CL
        ADD     DL,AL                   ;e somar a indentacao
        CALL    GOTO_XY                 ;para obter a coluna do cursor fantasma
        POP     DX
        POP     CX
        POP     AX
        RET
MOVE_POS_HEXA ENDP

        PUBLIC  MOVE_POS_ASCII
        EXTRN   GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   LINHAS_ANTES_SETOR:BYTE
DATA_SEG        ENDS

;-------------------------------------------------------------------------;
; Esta rotina move o cursor real para o inicio do cursor                  ;
; fantasma na janela ASCII.                                               ;
;                                                                         ;
; Usa:      GOTO_XY                                                       ;
; Le:       LINHAS_ANTES_SETOR, CURSOR_FANT_X, CURSOR_FANT_Y              ;
;-------------------------------------------------------------------------;

MOV_POS_ASCII   PROC    NEAR
        PUSH    AX
        PUSH    DX
        MOV     DH,LINHAS_ANTES_SETOR   ;Encontra linha do fantasma (0,0)
        ADD     DH,2                    ;Mais linha seguinte
        ADD     DH,CURSOR_FANT_Y        ;DH = linha do cursor fantasma
        MOV     DL,59                   ;Indenta a esquerda
        ADD     DL,CURSOR_FANT_X        ;Soma CURSOR_X para obter
        CALL    GOTO_XY                 ;a posicao do cursor fantasma
        POP     DX
        POP     AX
        RET
MOVE_POS_ASCII          ENDP

        PUBLIC  SALVA_CURSOR_REAL
```

```
;-------------------------------------------------------------------------;
; Esta rotina salva a posicao do cursor real                              ;
; nas duas variaveis abaixo:                                              ;
;                                                                         ;
; Escreve:      CURSOR_REAL_X, CURSOR_REAL_Y                              ;
;-------------------------------------------------------------------------;
SALVA_CURSOR_REAL      PROC      NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Le posicao do cursor
        XOR     BH,BH                   ;na pagina 0
        INT     10h                     ;E a retorna em DL,DH
        MOV     CURSOR_REAL_Y,DL        ;Salva posicao
        MOV     CURSOR_REAL_X,DH
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
SALVA_CURSOR_REAL    ENDP

        PUBLIC  RESTAURA_CURSOR_REAL
        EXTRN   GOTO_XY:NEAR

;-------------------------------------------------------------------------;
; Esta rotina restaura o cursor real a sua antiga posição,                ;
; salva pelas variáveis abaixo.                                           ;
;                                                                         ;
; Usa:      GOTO_XY                                                       ;
; Le:       CURSOR_REAL_X, CURSOR_REAL_Y                                  ;
;-------------------------------------------------------------------------;

RESTAURA_CURSOR_REAL      PROC      NEAR
        PUSH    DX
        MOV     DL,CURSOR_REAL_Y
        MOV     DH,CURSOR_REAL_X
        CALL    GOTO_XY
        POP     DX
        RET
RESTAURA_CURSOR_REAL      ENDP

        PUBLIC  IMP_FANTASMA
        EXTRN   IMP_ATRIBUTO_N_VEZES:NEAR

;-------------------------------------------------------------------------;
; Esta rotina usa CURSOR_X e CURSOR_Y, atraves de MOVE_POS_...,           ;
; como coordenadas para o cursor fantasma. IMP_FANTASMA imprime           ;
; o cursor fantasma.                                                      ;
;                                                                         ;
; Usa:      IMP_ATRIBUTO_N_VEZES, SALVA_CURSOR_REAL                       ;
;           RESTAURA_CURSOR_REAL, MOVE_POS_HEXA                           ;
;           MOVE_POS_ASCII                                                ;
;-------------------------------------------------------------------------;
IMP_FANTASMA    PROC      NEAR
        PUSH    CX
        PUSH    DX
        CALL    SALVA_CURSOR_REAL
        CALL    MOV_POS_HEXA            ;Coord. do cursor na janela hexa
        MOV     CX,4                    ;Cursor possui 4 caracteres de largura
        MOV     DL,70h
        CALL    IMP_ATRIBUTO_N_VEZES
        CALL    MOVE_POS_ASCII          ;Coord. do cursor na janela ASCII
        MOV     CX,1                    ;Cursor possui 1 caractere de largura
        CALL    IMP_ATRIBUTO_N_VEZES
        CALL    RESTAURA_CURSOR_REAL
        POP     DX
```

```
        POP     CX
        RET
IMP_FANTASMA    ENDP

;-------------------------------------------------------------------------
; Estas quatro rotinas movem os cursores fantasma.
;
; Usa:      APAGA_FANTASMA, IMP_FANTASMA
;           TELA_DESCE, TELA_SOBE
; Le:       CURSOR_FANT_X, CURSOR_FANT_Y
; Altera:   CURSOR_FANT_X, CURSOR_FANT_Y
;-------------------------------------------------------------------------
        PUBLIC  FANTASMA_CIMA
FANTASMA_CIMA   PROC    NEAR
        CALL    APAGA_FANTASMA          ;Apaga na posicao atual
        DEC     CURSOR_FANT_Y           ;Sobe cursor uma linha
        JNS     NAO_ESTAVA_TOPO         ;Nao estava no alto, imprime
        CALL    TELA_DESCE              ;Estava no alto, desce tela
NAO_ESTAVA_TOPO
        CALL    IMP_FANTASMA            ;Imprime o cursor na nova posicao
        RET
FANTASMA_CIMA   ENDP

        PUBLIC  FANTASMA_BAIXO
FANTASMA_BAIXO  PROC    NEAR
        CALL    APAGA_FANTASMA          ;Apaga na posicao atual
        INC     CURSOR_FANT_Y           ;Abaixa o cursor uma linha
        CMP     CURSOR_FANT_Y,16        ;Estava no fundo?
        JB      NAO_ESTAVA_FUNDO        ;Nao, imprime fantasma
        CALL    TELA_SOBE               ;Sim, sobe a tela
NAO_ESTAVA_FUNDO:
        CALL    IMP_FANTASMA            ;Imprime o cursor fantasma
        RET
FANTASMA_BAIXO  ENDP
        PUBLIC  FANTASMA_ESQ
FANTASMA_ESQ    PROC    NEAR
        CALL    APAGA_FANTASMA  ;Apaga na posicao atual
        DEC     CURSOR_FANT_X   ;Move uma coluna a esquerda
        JNS     NAO_ESTAVA_ESQ  ;Nao estava do lado direito, imprime cursor
        MOV     CURSOR_FANT_X,0 ;Estava na esquerda, coloca de volta
NAO_ESTAVA_ESQ
        CALL    IMP_FANTASMA    ;Imprime o cursor fantasma
        RET
FANTASMA_ESQ    ENDP

        PUBLIC  FANTASMA_DIR
FANTASMA_DIR    PROC    NEAR
        CALL    APAGA_FANTASMA  ;Apaga na posicao atual
        INC     CURSOR_FANT_X   ;Move uma coluna a direita
        CMP     CURSOR_FANT_X,16 ;Ja esta no lado direito?
        JB      NAO_ESTAVA_DIR
        MOV     CURSOR_FANT_X,15 ;Estava na direita, coloca de volta
NAO_ESTAVA_DIR:
        CALL    IMP_FANTASMA    ;Imprime o cursor fantasma
        RET
FANTASMA_DIR    ENDP

        EXTRN   DISP_MEIO_SETOR:NEAR, GOTO_XY:NEAR
DATA_SEG
                SEGMENT PUBLIC
        EXTRN   DESL_SETOR:WORD
        EXTRN   LINHAS_ANTES_SETOR:BYTE
                ENDS
```

```
;-------------------------------------------------------------------------;
; Estas duas rotinas fazem a troca entre as duas metades do setor.        ;
;                                                                         ;
; Usa:      IMP_FANTASMA, DISP_MEIO_SETOR, APAGA_FANTASMA, GOTO_XY        ;
;           SALVA_CURSOR_REAL, RESTAURA_CURSOR_REAL                       ;
; Le:       LINHAS_ANTES_SETOR                                            ;
; Escreve:  DESL_SETOR, CURSOR_FANT_Y                                     ;
;-------------------------------------------------------------------------;
TELA_SOBE       PROC    NEAR
        PUSH    DX
        CALL    APAGA_FANTASMA          ;Remove o cursor fantasma
        CALL    SALVA_CURSOR_REAL       ;Salva posicao do cursor real
        XOR     DL,DL                   ;Define cursor para tela de meio-setor
        MOV     DH,LINHAS_ANTES_SETOR
        ADD     DH,2
        CALL    GOTO_XY
        MOV     DX,256                  ;Apresenta a segunda metade do cursor
        MOV     DESL_SETOR,DX
        CALL    DISP_MEIO_SETOR
        CALL    RESTAURA_CURSOR_REAL    ;Restaura posicao do cursor real
        MOV     CURSOR_FANT_Y,0         ;Cursor no alto da segunda metade
        CALL    IMP_FANTASMA            ;Restaura o cursor fantasma
        POP     DX
        RET
TELA_SOBE       ENDP

TELA_DESCE      PROC    NEAR
        PUSH    DX
        CALL    APAGA_FANTASMA          ;Remove o cursor fantasma
        CALL    SALVA_CURSOR_REAL       ;Salva posicao do cursor
        XOR     DL,DL                   ;Define cursor para tela de meio-setor
        MOV     DH,LINHAS_ANTES_SETOR
        ADD     DH,2
        CALL    GOTO_XY
        XOR     DX,DX                   ;Apresenta a primeira metade do cursor
        MOV     DESL_SETOR,DX
        CALL    DISP_MEIO_SETOR
        CALL    RESTAURA_CURSOR_REAL    ;Restaura posicao do cursor real
        MOV     CURSOR_FANT_Y,15        ;Cursor no fundo da primeira metade
        CALL    IMP_FANTASMA            ;Restaura o cursor fantasma
        POP     DX
        RET
TELA_DESCE      ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
CURSOR_REAL_X           DB      0
CURSOR_REAL_Y           DB      0
        PUBLIC  CURSOR_FANT_X, CURSOR_FANT_Y
CURSOR_FANT_X           DB      0
CURSOR_FANT_Y           DB      0
DATA_SEG        ENDS
```

VIDEO_IO.ASM

```
CGROUP  GROUP   CODE_SEG, DATA_SEG
        ASSUME  CS:CGROUP, DS:CGROUP

CODE_SEG        SEGMENT PUBLIC
        ORG     100h

        PUBLIC  IMP_HEXA

;-------------------------------------------------------------------------;
; Esta rotina converte o byte em DL para hexa e imprime os              ;
; dois digitos hexa na posicao corrente do cursor.                      ;
;                                                                       ;
;       DL      Byte a ser convertido para hexa                         ;
; Usa:          IMP_DIG_HEXA                                            ;
;-------------------------------------------------------------------------;
IMP_HEXA        PROC    NEAR            ;Ponto de entrada
        PUSH    CX                      ;Salva registradores usados
        PUSH    DX                      ;   nesta rotina
        MOV     DH,DL                   ;Copia o byte
        MOV     CX,4                    ;Coloca nibble alto em DL
        SHR     DL,CL
        CALL    IMP_DIG_HEXA            ;Imprime primeiro digito
        MOV     DL,DH                   ;Nibble baixo para DL
        AND     DL,0Fh                  ;Remove nibble alto
        CALL    IMP_DIG_HEXA            ;Imprime segundo digito
        POP     DX
        POP     CX
        RET
IMP_HEXA        ENDP

        PUBLIC  IMP_DIG_HEXA

;-------------------------------------------------------------------------;
; Esta rotina converte os 4 bits baixos de DL para um digito            ;
; hexa e o imprime na tela.                                             ;
;                                                                       ;
;       DL      4 bits baixos contem numero a ser impresso              ;
;                                                                       ;
; Usa:          IMP_CAR                                                 ;
;-------------------------------------------------------------------------;
IMP_DIG_HEXA    PROC    NEAR
        PUSH    DX                      ;Salva registradores usados
        CMP     DL,10                   ;O nubble >10?
        JAE     LETRA_HEXA              ;Nao, converte para letra
        ADD     DL,"0"                  ;Sim, converte para digito
        JMP     Short IMP_DIGITO        ;Imprime este caractere
LETRA_HEXA:
        ADD     DL,"A"-10               ;Converte para letra hexa
IMP_DIGITO:
        CALL    IMP_CAR                 ;Imprime letra na tela
        POP     DX                      ;Restaura valor de DX
        RET
IMP_DIG_HEXA    ENDP

        PUBLIC  IMP_CAR
        EXTRN   CURSOR_DIREITA:NEAR

;-------------------------------------------------------------------------;
; Esta rotina envia um caractere para a tela usando as rotinas do       ;
; ROM BIOS, para que os caracteres especiais sejam testados e impressos ;
; como outro qualquer.                                                  ;
; Esta rotina deve realizar um certo trabalho, para mudar o cursor.     ;
;                                                                       ;
;   DL      Byte a imprimir na tela                                     ;
;                                                                       ;
; Usa:      CURSOR_DIREITA                                              ;
;-------------------------------------------------------------------------;
```

```
IMP_CAR         PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,9            ;Chamada para sair caractere/atributo
        MOV     BH,0            ;Define pagina 0
        MOV     CX,1            ;Imprime somente 1 caractere
        MOV     AL,DL           ;Caractere a imprimir
        MOV     BL,7            ;Atributo normal
        INT     10h             ;Imprime caractere e atributo
        CALL    CURSOR_DIRETA   ;Move agora a proxima posicao do cursor
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
IMP_CAR         ENDP

        PUBLIC  IMP_DECIMAL

;-------------------------------------------------------------------------
; Esta rotina imprime um numero de 16 bits sem sinal em                  ;
; notacao decimal.                                                       ;
;                                                                        ;
;       DX      N : numero de 16 bits sem sinal                          ;
;                                                                        ;
; Usa:          IMP_DIG_HEXA                                             ;
;-------------------------------------------------------------------------
IMP_DECIMAL
        PUSH    AX              ;Salva registradores usados
        PUSH    CX
        PUSH    DX
        PUSH    SI
        MOV     AX,DX
        MOV     SI,10           ;Dividira por 10 usando SI
        XOR     CX,CX           ;Contador digitos na pilha
NAO_ZERO:
        XOR     DX,DX           ;Zera palavra de N
        DIV     SI              ;Calcula N/10 e (N mod 10)
        PUSH    DX              ;Coloca um digito na pilha
        INC     CX              ;Mais um digito colocado
        OR      AX,AX           ;N ja igual a 0?
        JNE     NAO_ZERO        ;Nao, continua.
IMP_DIG_LOOP:
        POP     DX              ;Apanha os digitos em
                                ;     ordem contraria
        CALL    IMP_DIG_HEXA
        LOOP    IMP_DIG_LOOP
FIM_DECIMAL:
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
IMP_DECIMAL     ENDP

        PUBLIC  IMP_CAR_N VEZES

;-------------------------------------------------------------------------
; Esta rotina imprime mais de uma copia de um caractere                  ;
;                                                                        ;
;    DL     Codigo do caracter                                          ;
;    CX     Numero de vezes a imprimir                                  ;
;                                                                        ;
; Usa:        IMP_CAR                                                  ;
;-------------------------------------------------------------------------
IMP_CAR_N_VEZES         PROC    NEAR
```

```
        PUSH    CX
N_VEZES:
        CALL    IMP_CAR
        LOOP    N_VEZES
        POP     CX
        RET
IMP_CAR_N_VEZES         ENDP

        PUBLIC  IMP_FORMA

;-------------------------------------------------------------------------;
; Esta rotina imprime uma linha na tela, baseada nos dados              ;
; de formato                                                            ;
;                                                                       ;
;       DB      caractere, numero de vezes a imprimir   ,0              ;
;         x     significa que x pode ser repetido qualquer numero de vezes
;       DB      Endereco do comando de dados acima.                     ;
;                                                                       ;
; Usa:          IMP_CAR_N_VEZES                                         ;
;-------------------------------------------------------------------------;
IMP_FORMA       PROC    NEAR
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSHF                           ;Salva o flag de direcao
        CLD                             ;Seta o flag direcao p/incremento
        MOV     SI,DX                   ;Deslocamento --> SI para LODSB
LOOP_FORMA:
        LODSB                           ;Pega o dado e coloca em AL
        OR      AL,AL                   ;E o fim da lista (0h)?
        JZ      AL,AL                   ;Sim, retorna
        MOV     DL,AL                   ;Nao, prepara para imprimir N vezes
        LODSB                           ;Apanha o contador para AL
        MOV     CL,AL                   ;Coloca em CX para IMP_CAR_N_VEZES
        XOR     CH,CH                   ;Zera byte alto de CX
        CALL    IMP_CAR_N_VEZES
        JMP     LOOP_FORMA
FIM_FORMA:
        POPF                            ;Restaura flag de direcao
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
IMP_FORMA       ENDP

        PUBLIC  IMP_CAB
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_LINHA_CAB:BYTE
        EXTRN   CAB_1:BYTE
        EXTRN   CAB_2:BYTE
        EXTRN   NR_DRIVE:BYTE
        EXTRN   NR_SETOR:WORD
DATA_SEG        ENDS
        EXTRN   GOTO_XY:NEAR, LIMPA_ATE_FINAL_LINHA:NEAR

;-------------------------------------------------------------------------;
; Esta rotina imprime o cabecalho com drive e numero do setor.          ;
;                                                                       ;
; Usa:      GOTO_XY, IMP_STRING, IMP_CAR, IMP_DECIMAL                   ;
;           LIMPA_ATE_FINAL_LINHA                                       ;
; Le:       NR_LINHA_CAB, CAB_1, CAB_2                                  ;
;           NR_DRIVE, NR_SETOR                                          ;
;-------------------------------------------------------------------------;
IMP_CAB         PROC    NEAR
        PUSH    DX
        XOR     DL,DL                   ;Move cursor para numero de linha do cab
```

```
        MOV     NR_LINHA_CAB
        CALL    GOTO_XY
        LEA     DX,CAB_1
        CALL    IMP_STRING
        MOV     DL,NR_DRIVE
        ADD     DL,'A'                  Imprime drives A, B, ...
        CALL    IMP_CAR
        LEA     DX,CAB_2
        CALL    IMP_STRING
        MOV     DX,NR_SETOR
        CALL    IMP_DECIMAL
        CALL    LIMPA_ATE_FINAL_LINHA   ;Limpa restante do numero do setor
        POP     DX
        RET
IMP_CAB ENDP

        PUBLIC  IMP_STRING

;-------------------------------------------------------------------------
; Esta rotina imprime um string de caracteres na tela.                   ;
; O string deve terminar com: DB    0                                    ;
;                                                                        ;
;       DS:BX   Endereço do string                                       ;
;                                                                        ;
; Usa:          IMP_CAR                                                  ;
;-------------------------------------------------------------------------
IMP_STRING      PROC    NEAR
        PUSH    AX
        PUSH    DX
        PUSH    SI
        PUSHF                           ;Salva flag direcao
        CLD                             ;Seta direcao p/incremento
        MOV     SI,DX                   ;Coloca endereço em SI para LODSB
LOOP_STRING:
        LODSB                           ;Coloca caractere em AL
        OR      AL,AL                   ;Ja encontrou o 0?
        JZ      FIM_STRING              ;Sim, terminou o string
        MOV     DL,AL                   ;Nao, imprime caractere
        CALL    IMP_CAR
        JMP     LOOP_STRING
FIM_STRING:
        POPF                            ;Restaura flag de direcao
        POP     SI
        POP     DX
        POP     AX
        RET
IMP_STRING      ENDP

        PUBLIC  IMP_PRONTO
        EXTRN   LIMPA_ATE_FINAL_LINHA:NEAR
        EXTRN   GOTO_XY:NEAR
DATA_SEG        SEGMENT PUBLIC
        EXTRN   NR_LINHA_PRONTO:BYTE
DATA_SEG        ENDS

;-------------------------------------------------------------------------
; Esta rotina imprime a linha de pronto na tela e limpa ate o            ;
; fim da linha                                                           ;
;                                                                        ;
;       DS:DX   Endereço da mensagem                                     ;
;                                                                        ;
; Usa:          IMP_STRING, LIMPA_ATE_FINAL_LINHA, GOTO_XY               ;
; Le:           NR_LINHA_PRONTO                                          ;
;-------------------------------------------------------------------------

IMP_PRONTO      PROC    NEAR
        PUSH    DX
        XOR     DL,DL                   ;Imprime a linha de pronto e
```

```
        MOV     DH,NR_LINHA_PRONTO          ;move o cursor
        CALL    GOTO_XY
        POP     DX
        CALL    IMP_STRING
        CALL    LIMPA_ATE_FINAL_LINHA
        RET
IMP_PRONTO          ENDP

        PUBLIC  IMP_ATRIBUTO_N_VEZES
        EXTRN   CURSOR_DIREITA:NEAR

;-------------------------------------------------------------------------
; Esta rotina coloca o atributo em N caracteres, começando
; na posicao atual do cursor.
;
;     CX      Numero de caracteres colocar o atributo
;     DL      Novo atributo dos caracteres
;
; Usa:        CURSOR_DIREITA
;-------------------------------------------------------------------------

IMP_ATRIBUTO_N_VEZES        PROC    NEAR
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BL,DL               ;Seta novo atributo
        XOR     BH,BH               ;Pagina de video = 0
        MOV     DX,CX               ;CX e usado pelo ROM BIOS
        MOV     CX,1                ;Seta atributo para um caractere
LOOP_ATR:
        MOV     AH,8                ;Le o caractere sob o cursor
        INT     10h
        MOV     AH,9                ;Imprime atributo/caractere
        INT     10h
        CALL    CURSOR_DIREITA
        DEC     DX                  ;Ja imprimiu N caracteres?
        JNZ     LOOP_ATR            ;Nao, continua
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
IMP_ATRIBUTO_N_VEZES        ENDP

CODE_SEG        ENDS

        END
```

APÊNDICE B

ORDEM DE CARGA DOS SEGMENTOS

Ordem de Carga dos Segmentos

Erros de Fase

Arquivo EXE2BIN Não Pode Ser Convertido

O Macro Assembler da IBM (versões 1.0 e 1.2) carrega os segmentos na memória em uma ordem diferente da ordem usada por todas as versões mais recentes do Macro Assembler da Microsoft. Neste Apêndice, veremos a questão da ordem de carga dos segmentos, e também como esse conhecimento pode ser útil a você quando o EXE2BIN apresenta a mensagem de erro *File cannot be converted*.

Ordem de Carga dos Segmentos

Todos os exemplos após o Capítulo 13 usam dois segmentos, CODE_SEG e DATA_SEG. As versões IBM do Macro Assembler dizem ao LINK para carregar esses segmentos na memória em ordem alfabética. Assim, quando escrevermos:

```
CGROUP   GROUP  CODE_SEG, DATA_SEG
         ASSUME CS:CGROUP, DS:CGROUP
DATA_SEG        SEGMENT PUBLIC
DATA_SEG        ENDS

CODE_SEG        SEGMENT PUBLIC
CODE_SEG        ENDS

         END
```

as versões IBM do Macro Assembler dirão ao LINK para carregar DATA_SEG para a memória após ter carregado CODE_SEG. Vamos passar esse fragmento de código para um programa real, a fim de que possamos ver o mapa de carga.

Eis nossa nova versão. Ela não faz muita coisa, mas é o suficiente para vermos como o LINK carrega os segmentos para a memória.

```
CGROUP   GROUP    CODE_SEG, DATA_SEG
         ASSUME   CS:CGROUP, DS:CGROUP

DATA_SEG         SEGMENT PUBLIC
         DB      0
DATA_SEG         ENDS

CODE_SEG         SEGMENT PUBLIC
         ORG     100h
INICIO:  INT     20h
CODE_SEG         ENDS

         END     INICIO
```

Digite esse arquivo, e chame-o de SEGTESTE.ASM, e depois monte-o e linkedite-o para criar o mapa de carga:

```
A>LINK SEGTESTE,SEGTESTE,SEGTESTE/MAP
```

Se você tiver uma versão IBM do assembler, você verá esse mapa de carga:

```
Warning: no stack segment

Start  Stop   Length Name              Class
00000H 00101H 00102H CODE_SEG
00110H 00110H 00001H DATA_SEG

Origin         Group
0000:0         CGROUP

Address              Publics by Name

Address              Publics by Value

Program entry point at 0000:0100
```

LINK carregou CODE_SEG para a memória antes de DATA_SEG. Essa é exatamente a ordem que queremos. De fato, CODE_SEG *deve* ser o primeiro segmento na memória, de modo que nosso programa comece em 100h a partir do início do grupo.

Por outro lado, seu mapa pode indicar que os dois segmentos foram carregados em ordem contrária. Isso indica que você tem uma versão do assembler da Microsoft. Se for isso, você verá um mapa de carga semelhante a esse:

```
Warning: no stack segment

Start  Stop   Length  Name            Class
00000H 00000H 00001H DATA_SEG
00010H 00111H 00102H CODE_SEG

Origin         Group
0000:0         CGROUP

Address              Publics by Name

Address              Publics by Value

Program entry point at 0000:0110
```

Nada está certo nesse mapa de carga. DATA_SEG aparece na memória antes de CODE_SEG, e isso significa que o comando ORG 100h nos dá um deslocamento a partir do final do segmento de dados, e não do início do grupo.

A última linha do mapa mostra que o endereço de entrada do nosso arquivo é 110h. Mas ele precisa ser 100H para um arquivo .COM. Assim, o que acontecerá se tentarmos criar um arquivo .COM com essa ordem?

Execute EXE2BIN e você verá o seguinte:

```
A>EXE2BIN SEGTESTE SEGTESTE.COM
File cannot be converted
A>
```

Essa não é uma mensagem de erro muito útil — ela não nos dá uma dica sobre o tipo de erro que ocasionou essa impossibilidade. Mas é nessa hora que o mapa de carga se torna útil. Olhando para o mapa de carga, podemos ver que LINK carregou os segmentos na memória na ordem errada. Depois, só temos que pensar em um modo de resolver o problema.

Tivemos o cuidado neste livro de fazer com que todos os programas pudessem ser executados nas duas versões do assembler — da Microsoft e da IBM. É por isso que colocamos o segmento de dados *depois* do segmento de código em todos os nossos arquivos fonte.

Se, no seu trabalho com os programas em linguagem assembly, você criou ou encontrou programas em que o segmento de dados aparece no alto do arquivo, use a chave */A* disponível com as versões Microsoft do MASM. A opção */A* diz ao MASM que você quer os segmentos carregados em ordem alfabética. Para experimentar essa opção, monte o nosso programa de teste, SEGTESTE.ASM, com o seguinte comando:

```
A>MASM SEGTESTE /A;
```

Linkedite novamente esse arquivo e crie o mapa de carga. Você deverá ver os dois segmentos em ordem alfabética, com o CODE_SEG em primeiro lugar no arquivo.

Erros de Fase

Tivemos o cuidado de ter todos os nossos exemplos executáveis em todas as versões do Macro Assembler — IBM e Microsoft — colocando o segmento de dados ao final dos arquivos. Mas

essa não é uma boa idéia em muitos casos. Nesta seção, vamos dar uma olhada nesse problema, e descobrir os melhores modos de organizar seus segmentos.

Vejamos um exemplo concreto:

```
CODE_SEG        SEGMENT PUBLIC
        ASSUME  CS:CODE_SEG, ES:DATA_SEG

BEGIN   PROC    NEAR
        MOV     AX,DATA_SEG             ;Apanha numero do segmento
        MOV     ES,AX                   ;Define ES para que aponte para dados
        MOV     AL,VARIAVEL             ;Le variavel em AL
        MOV     AH,4Ch                  ;Saida para DOS
        INT     21h
BEGIN   ENDP

CODE_SEG        ENDS

DATA_SEG        SEGMENT PUBLIC
VARIAVEL        DB      0
DATA_SEG        ENDS

        END     BEGIN
```

Colocamos o segmento de dados no final desse programa para termos certeza de que DATA_SEG seja carregado depois de CODE_SEG. Mas o assembler gera uma mensagem de *erro de fase* quando tenta montá-lo:

```
A>MASM TESTE;
Microsoft (R) Macro Assembler   Version 4.00
Copyright (C) Microsoft Corp 1981, 1983, 1984, 1985. All rights reserved.

TESTE.ASM(10) : error b: Phase error between passes

    51036 Bytes symbol space free

        0 Warning Errors
        1 Severe
        1         Errors
```

O que significa erro de fase?

Acontece que o Macro Assembler faz diversas passagens pelo arquivo conforme o está montando. Na primeira passagem, ele apanha a informação de que precisa, tal como o tipo e os segmentos das variáveis. Por questão de eficiência, o assembler também começa a montar o programa na primeira passagem; e aí surgem os problemas.

O MASM monta a instrução MOV AL,VARIAVEL *antes* de saber qual o segmento que contém VARIAVEL, de modo que a instrução MOV é montada como se não precisássemos de uma superposição de segmento (que é ES: nesse caso). Na segunda passagem, no entanto, o MASM nota que precisa acrescentar esse "prefixo" pois VARIAVEL está no segmento apontado por ES. Infelizmente, o MASM não reservou espaço para isso durante a primeira passagem (ou fase), de modo que ele gera uma mensagem de *erro de fase*.

Precisamos declarar todas as variáveis *antes* de as usarmos em um arquivo. Se fizermos isso, e estivermos usando o Macro Assembler da Microsoft, o segmento de dados será colocado na memória em primeiro lugar, que geralmente não é problema com arquivos .EXE, onde a maioria provavelmente usa múltiplos segmentos.

Se, por outro lado, você quiser que o segmento de código seja carregado em primeiro lugar na memória, há uma solução simples: coloque apenas um segmento *fictício* antes do seu segmento de dados. Você pode ver os detalhes no exemplo a seguir.

```
CODE__SEG       SEGMENT PUBLIC          ;Carrega CODE__SEG primeiro
CODE__SEG       ENDS

DATA__SEG       SEGMENT PUBLIC
VARIAVEL        DB      0
DATA__SEG       ENDS

CODE__SEG       SEGMENT PUBLIC
        ASSUME  CS:CODE__SEG, ES:DATA__SEG

BEGIN   PROC    NEAR
        MOV     AX,DATA__SEG        ;Apanha numero do segmento
        MOV     ES,AX               ;Define ES para que aponte para dados
        MOV     AL,VARIAVEL         ;Le variavel em AL
        MOV     AH,4Ch              ;Saida para DOS
        INT     21h
BEGIN   ENDP

CODE__SEG       ENDS

        END     BEGIN
```

Arquivo EXE2BIN Não Pode Ser Convertido

Se você tiver problemas com EXE2BIN, primeiro verifique se no mapa de carga o segmento de código é o primeiro segmento. Além disso, assegure-se de ter somente dois segmentos listados. É possível ter diversas versões diferentes do mesmo segmento listado. Por exemplo:

```
00000H 00103H 00104H CODE__SEG
00110H 00105H 00006H DATA__SEG
00120H 00101H 00102H CODE__SEG
```

Nesse caso, CODE__SEG está fragmentado. Se você tiver mais de um trecho do mesmo segmento no mapa de carga, isso significa que você tem problemas, e eles podem indicar diversos problemas possíveis.

- Você pode não ter uma pseudo-op PUBLIC após todas as suas definições de segmento.
- Você pode ter definições SEGMENT ligeiramente diferentes em seus arquivos fonte. Verifique em todos os seus arquivos fonte se todas as definições SEGMENT são idênticas.
- Um dos seus arquivos fonte pode não ter um comando GROUP ou o comando GROUP pode não estar correto. Verifique cuidadosamente todos os comandos GROUP, e veja se eles são idênticos.
- Se os comandos GROUP estiverem em ordem, verifique se os comandos ASSUME estão da seguinte forma:

```
ASSUME CS:CGROUP, DS:CGROUP
```

  se você usou um nome de grupo CGROUP.
- Você definiu um segmento de pilha. Os programas .COM não precisam de um segmento de pilha e, na verdade, você *não pode* definir um.
- O ponto de entrada não está em 100h. Isso pode ser porque você não colocou o nome da rotina inicial após a pseudo-op END no arquivo fonte principal, ou linkeditou os arquivos em ordem errada. A rotina principal *deve* ser o primeiro nome de arquivo na lista do LINK.

Você também encontrará mais informações sobre mensagens de erro e o que elas significam no Apêndice C.

APÊNDICE C

MENSAGENS DE ERRO COMUNS

MASM

LINK

EXE2BIN

Este apêndice lista muitas das mensagens de erro comuns que você pode encontrar quando estiver usando o MASM, LINK e EXE2BIN. Se você não encontrar uma mensagem de erro aqui, verifique no seu manual do macro assembler ou do DOS.

As mensagens de erro estão em três grupos: um para o MASM, um para o LINK, e outro para o EXE2BIN. Dentro de cada seção, as mensagens estão listadas em ordem alfabética.

MASM

Block nesting error — Provavelmente essa mensagem aparecerá junto com as mensagens *Open procedures* ou *Open Segments*. Veja as descrições para essas duas mensagens.

End of file, no END directive — Ou você não colocou o comando END no final do arquivo, ou precisa colocar uma linha em branco após o comando END já existente. As versões Microsoft do macro assembler esperam encontrar uma linha em branco ao final do arquivo. Se você não tiver pelo menos uma linha em branco após o END, o MASM não lerá o comando END.

Must be declared in pass 1 — Essa mensagem geralmente aparece em conjunto com o comando GROUP. Ela significa que você não definiu um dos segmentos listados no comando GROUP. Por exemplo, se você tiver a linha *CGROUP GROUP CODE_SEG, DATA_SEG*, mas nunca definiu um segmento chamado DATA_SEG, provavelmente verá essa mensagem. Verifique se você declarou todos os segmentos listados no comando GROUP.

No or unreachable CS — O MASM precisa ver um comando ASSUME a fim de saber como montar algumas instruções, como instruções de desvio e chamadas. Essa mensagem significa ou que o MASM não encontrou um comando ASSUME ou que o ASSUME que ele encontrou tinha um erro. Verifique se existe o comando ASSUME em seu arquivo fonte, e que o comando está correto.

Open procedures — Isso significa que está faltando ou um comando PROC ou um ENDP, ou que os nomes não são os mesmos no par PROC/ENDP. Assegure-se de que todo PROC combina com um ENDP, e que possuem o mesmo nome.

Open segments — Falta um comando SEGMENT ou ENDS, ou os nomes não são os mesmos no par SEGMENT/ENDS. Assegure-se de que todo comando SEGMENT combina com um ENDS, e que possuem o mesmo nome.

Symbol not defined — Há três coisas que você deve verificar se encontrar essa mensagem de erro:

1. Você pode ter escrito um nome errado. Verifique a linha em que você viu o erro para ter certeza de que o nome está correto.
2. Você pode ter errado no nome quando declarou um PROC ou uma variável. Verifique se os nomes da linha onde houve o erro combinam com suas declarações de rotinas e variáveis.
3. Pode estar faltando uma declaração EXTRN, ou o nome em EXTRN pode estar errado.

LINK

Fixup offset exceeds width — Esse, provavelmente, é o erro mais difícil de se matar. Essa mensagem geralmente significa que você declarou alguma rotina como FAR, e depois declarou a mesma rotina como NEAR em uma declaração EXTRN.

Também pode significar que um grupo tornou-se maior do que os 64K permitidos para os grupos. Você pode verificar isso pelo campo de tamanho no arquivo de mapa.

Essa mensagem também pode aparecer quando o seu segmento foi fragmentado. Nesse caso, os dois fragmentos podem estar mais de 64K separados, o que significa que os CALLs devem ser FAR para que funcionem. Você verá mais sobre segmentos fragmentados no Apêndice B.

Se isso não parece ser o problema, você terá que examinar mais a fundo. Leia o Apêndice B cuidadosamente, e depois crie um mapa de carga do seu programa. Você pode descobrir uma dica por esse mapa de carga. Por exemplo, verifique a ordem dos segmentos. Você pode descobrir que eles estão fora de ordem.

Simbol defined more than once — Isso provavelmente significa que você definiu a mesma rotina ou variável em dois arquivos fonte. Assegure-se de ter definido cada nome em somente um arquivo fonte, e depois usado EXTRNs em outros lugares onde precisar usar a mesma rotina ou variável.

Unresolved externals — Quando você vir essa mensagem, ou está faltando um PUBLIC no arquivo em que você declarou a rotina ou variável, ou errou no nome em uma declaração EXTRN em algum outro arquivo fonte.

Warning: no stack segment — Essa não é exatamente uma mensagem de erro, é simplesmente um aviso. Você verá essa mensagem nos exemplos deste livro, pois estamos criando arquivos .COM, e os arquivos .COM não usam um segmento separado para a pilha. Veja o Capítulo 28 para um exemplo de um programa que não faz com que o LINK apresente esse aviso.

EXE2BIN

File cannot be converted — Essa provavelmente será a única mensagem de erro que você verá de EXE2BIN, e ela não ajuda muito. Na maioria das vezes ela significa uma dessas três coisas:

1. Seus segmentos estão na ordem errada, e dessa forma você terá um segmento na memória antes do segmento de código. Verifique o mapa de carga para ver se esse é o problema. Para mais informação, leia o Apêndice B.
2. Seu arquivo principal não é o primeiro arquivo listado no comando LINK. Ele deve ser o primeiro; tente linkeditar novamente para ver se o problema não foi esse. Novamente, você pode verificar esse tipo de problema pelo mapa de carga.
3. O seu programa principal não possui um comando ORG 100h como primeiro comando após *CODE_SEG SEGMENT PUBLIC*. Além disso, assegure-se de que o comando END em seu arquivo fonte principal inclui o rótulo da instrução em que você quer iniciar — por exemplo, *END DSKPATCH*.

Se essas sugestões não o ajudarem, verifique outras informações no Apêndice B.

APÊNDICE D

DIVERSAS TABELAS

Códigos de Caracteres ASCII

Códigos Estendidos do Teclado

Tabela de Modos de Endereçamento

Funções de INT 10h

Funções de INT 21h

TABELA D-1

CÓDIGOS DE CARACTERES ASCII

Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car
Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car
0	0		43	2B	+	86	56	V	129	81	ü
1	1		44	2C	,	87	57	W	130	82	é
2	2		45	2D	−	88	58	X	131	83	â
3	3	♥	46	2E	.	89	59	Y	132	84	a
4	4	♦	47	2F	/	90	5A	Z	133	85	a
5	5	♣	48	30	0	91	5B	[	134	86	å
6	6	♠	49	31	2	92	5C	\	135	87	ç
7	7	•	50	32	3	93	5D	]	136	88	ê
8	8		51	33	4	94	5E	^	137	89	e
9	9	○	52	34	5	95	5F		138	8A	e`
10	A		53	35	6	96	60	`	139	8B	i
11	B	♂	54	36	7	97	61	a	140	8C	i^
12	C	♀	55	37	8	98	62	b	141	8D	i
13	D		56	38	9	99	63	c	142	8E	A
14	E		57	39	:	100	64	d	143	8F	Å
15	F	☼	58	3A	;	101	65	e	144	90	É
16	0	►	59	3B	<	102	66	f	145	91	æ
17	11	◄	60	3C	=	103	67	g	146	92	Æ
18	12		61	3D	>	104	68	h	147	93	ô
19	13	‼	62	3E	?	105	69	i	148	94	o
20	14	¶	63	3F	@	106	6A	j	149	95	o`
21	15	§	64	40	A	107	6B	k	150	96	û
22	16	▬	65	41	B	108	6C	l	151	97	ù
23	17		66	42	C	109	6D	m	152	98	y
24	18		67	43	D	110	6E	n	153	99	O
25	19		68	44	E	111	6F	o	154	9A	Ü
26	1A		69	45	F	112	70	p	155	9B	¢
27	1B		70	46	G	113	71	q	156	9C	£
28	1C		71	47	H	114	72	r	157	9D	
29	1D		72	48	I	115	73	s	158	9E	
30	1E	▲	73	49	J	116	74	t	159	9F	
31	1F	▼	74	4A	K	117	75	u	160	A0	á
32	20		75	4B	L	118	76	v	161	A1	í
33	1	!	76	4C	M	119	77	w	162	A2	ó
34	22	"	77	4D	N	120	78	x	163	A3	ú
35	23	#	78	4E	O	121	79	y	164	A4	ñ
36	24	$	79	4F	P	122	7A	z	165	A5	Ñ
37	25	%	80	50	Q	123	7B		166	A6	ª
38	26	&	81	51	R	124	7C	|	167	A7	º
39	27	'	82	52	S	125	7D		168	A8	¿
40	28	(	83	53	T	126	7E	~	169	A9	
41	29	)	84	54	U	127	7F		170	AA	
42	2A	*	85	55		128	80	Ç	171	AB	½
172	AC	¼	193	C1		214	D6		235	EB	δ
173	AD	¡	194	C2		215	D7		236	EC	
174	AE	«	195	C3		216	D8		237	ED	ø

Tabela D-1. *Continuação*

Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car
Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car	Dec	Hexa	Car
175	AF	»	196	C4		217	D9		238	EE	ϵ
176	B0		197	C5		218	DA		239	EF	
177	B1		198	C6		219	DB		240	F0	≡
178	B2		199	C7		220	DC		241	F1	±
179	B3		200	C8		221	DD		242	F2	≥
180	B4		201	C9		222	DE		243	F3	≤
181	B5		202	CA		223	DF		244	F4	
182	B6		203	CB		224	E0	α	245	F5	
183	B7		204	CC		225	E1	β	246	F6	÷
184	B8		205	CD		226	E2	Γ	247	F7	≈
185	B9		206	CE		227	E3	Π	248	F8	
186	BA		207	CF		228	E4	Σ	249	F9	
187	BB		208	D0		229	E5	σ	250	FA	
188	BC		209	D1		230	E6		251	FB	
189	BD		210	D2		231	E7		252	FC	
190	BE		211	D3		232	E8		253	FD	
191	BF		212	D4		233	E9		254	FE	
192	C0		213	D5		234	EA		255	FF	

Muitas das teclas no teclado (como as teclas de função) retornam um código de dois caracteres quando você lê as teclas por meio do DOS: um 0 decimal seguido pelo código de varredura. A tabela a seguir mostra os códigos de varredura para todas as teclas que não possuem código ASCII equivalente.

TABELA D-2

CÓDIGOS ESTENDIDOS DO TECLADO

15	Shift Tab
16-25	Teclas Alt para Q, W, E, R, T, Y, U, I, O, P
30-38	Teclas Alt para A, S, D, F, G, H, J, K, L
44-50	Teclas Alt para Z, X, C, V, B, N, M
59-68	F1 até F10
71	Home
72	Cursor para Cima
73	PgUp
75	Cursor Esquerda
77	Cursor Direita
79	End
80	Cursor para Baixo
81	PgDn
82	Ins
83	Del
84-93	Shift F1 até F10
94-103	Ctrl F1 até F10
104-113	Alt F1 até F10
114	Ctrl PrtSc
115	Ctrl Cursor-Esq
116	Ctrl Cursor-Dir
117	Ctrl End
118	Ctrl PgDn
119	Ctrl Home
120-131	Ctrl Alt para 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, ', =
132	Ctrl PgUp

TABELA D-3

OS MODOS DE ENDEREÇAMENTO

Modo de Endereçamento	Formato do Endereço	Registrador de Segmento Usado
Registrador	registrador (como AX)	Nenhum
Imediato	dado (como 12345)	Nenhum
	Modos de Endereçamento da Memória	
Registrador Indireto	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Relativo à Base *	rótulo[BX]	DS
	rótulo[BP]	SS
Indexado Direto *	rótulo[DI]	DS
	rótulo[SI]	DS
Indexado na Base *	rótulo[BX + SI]	DS
	rótulo[BX + DI]	DS
	rótulo[BP + SI]	SS
	rótulo[BP + DI]	SS
Comandos de String: (MOVSW, LODSB, *etc.*)		Lidos de DS:SI Escritos em ES:DI

* Rótulo[...] pode ser trocado por [desl + ...], onde "desl" é um deslocamento. Assim poderíamos escrever [10 + BX] e o endereço seria 10 + BX.

TABELA D-4

FUNÇÕES DE INT 10h

(AH)=0 **Seta o modo de vídeo.** O registrador AL contém o número do modo.

MODO TEXTO

(AL)=0	40 por 25, modo preto e branco
(AL)=1	40 por 25, colorido
(AL)=2	80 por 25, preto e branco
(AL)=3	80 por 25, colorido
(AL)=7	80 por 25, adaptador de vídeo monocromático

MODO GRÁFICO

(AL)=4	320 por 200, colorido
(AL)=5	320 por 320, preto e branco
(AL)=6	640 por 200, preto e branco

(AH)=2 **Seta a posição do cursor.**

(DH,DL)	Linha, coluna da nova posição do cursor; o canto superior esquerdo é (0,0).
(BH)	Número da página. Esse é o número da página de vídeo. O adaptador de gráficos coloridos possui espaço para diversas páginas de vídeo, mas a maioria dos programas usa a página 0.

(AH)=3 **Seta a posição do cursor.**

(BH)	Número da página.
Saída	(DH,DL) Linha, coluna do cursor (CH,CL) Tamanho do cursor

(AH)=4 **Lê a posição da caneta ótica.**
(Veja o Manual de Referência Técnica.)

(AH)=5 **Seleciona página de vídeo ativa.**

(AL)	Novo número de página (de 0 a 7 para os modos de 0 a 1; de 0 a 3 para modos 2 e 3).

(AH)=6 **Rolamento de tela.**

(AL)	Número de linhas a colocar no final da janela. Normalmente se coloca uma linha. Coloque zero para limpar a janela inteira.
(CH,CL)	Linha, coluna do canto superior esquerdo da janela.

(DH,DL) Linha, coluna do canto inferior direito da janela.
(BH) Atributo de vídeo a usar para linhas em branco.

(AH)=7 **Abaixamento da tela.**

Igual à função de rolamento (função 6), mas as linhas são deixadas em branco no alto.

(AH)=8 **Lê atributo e caractere na posição do cursor.**

(BH) Página de vídeo (somente modo texto)
(AL) Caractere lido
(AH) Atributo do caractere lido (somente modo texto)

(AH)=9 **Imprime atributo e caractere na posição do cursor.**

(BX) Página de vídeo (somente modo texto)
(CX) Número de vezes a imprimir o caractere e o atributo na tela
(AL) Caractere a imprimir
(BL) Atributo a imprimir

(AH)=10 **Imprime caractere na posição do cursor** (com atributo normal).

(BH) Página de vídeo
(CX) Número de vezes a imprimir o caractere
(AL) Caractere a imprimir

(AH)=11 a 13 **Diversas funções gráficas.** (Veja os detalhes no Manual de Referência Técnica)

(AH) = 14 **Imprime teletipo.** Imprime um caractere na tela e move o cursor para a próxima posição.

(AL) Caractere a imprimir
(BL) Cor do caractere (somente modo gráfico)
(BH) Página de vídeo (modo texto)

(AH)=15 **Retorna estado atual do vídeo.**

(AL) Modo de vídeo atualmente setado
(AH) Número de caracteres por linha
(BH) Páginas de vídeo ativas

A tabela a seguir contém as funções de INT 21h usadas neste livro. Para uma lista mais completa, você deve comprar o *Manual de Referência Técnica do DOS*, da IBM.

TABELA D-5

FUNÇÕES DE INT 21h

(AH)=1 **Entrada do teclado.** Essa função espera a digitação de um caractere no teclado. Ela repete o caractere na tela e retorna o código ASCII no registrador AL. Para os códigos estendidos do teclado, essa função retorna dois caracteres: um 0 em ASCII seguido pelo código de varredura (veja Tabela D-2).

(AL) Caractere lido do teclado

(AH)=2 **Saída na tela.** Apresenta um caractere na tela. Diversos caracteres possuem significado especial para essa função:

7 Bipe: envia um sinal sonoro de um segundo para o auto-falante.
8 Backpace: move o cursor uma posição à esquerda.
9 Tab: Move para a próxima parada de tabulação. As paradas de tabulação são marcadas a cada 8 caracteres.
0Ah Mudança de linha: Move o cursor para a próxima linha.
0Dh Retorno de carro: Move para o início da linha corrente.
(DL) Caractere a ser apresentado na tela.

(AH)=8 **Entrada de teclado sem eco.** Lê um caractere do teclado, mas não apresenta o caractere na tela.

(AL) Caractere lido do teclado.

(AH)=9 **Apresenta string.** Apresenta na tela o string apontado pelo par de registradores DS:DX. Você deve marcar o final do string com o caractere $.

DS:DX Aponta para o string a apresentar.

(AH)=0Ah **Lê string.** Lê um string pelo teclado. Veja Capítulo 23 para mais detalhes.

(AH)=4Ch **Saída para o DOS.** Retorna ao DOS, como INT 20h, mas funciona para programas .COM e .EXE. A função INT 20h usada neste livro só funciona para programas .COM.

(AL) Código de retorno. Normalmente é zero, mas você pode usar qualquer número e usar os comandos de lote do DOS (IF e ERRORLEVEL) para detectar erros.

ÍNDICE REMISSIVO

A

A, comando do Debug, **30-31**
ADC, **37**
ADD, **19, 20**
Adicionando um, INC, **58-59**
Adição com carry, **37**
Adição, comando hexaritmético, **5**
Endereços, **19**
  CALL e segmentos, **96**
  Par CS:IP, **90**
  efetivos e LEA, **129**
  instruções de interrupção, vetores, **99**
  rótulos, **73-74**
  posições na memória, **18-19**
  arquivos de mapa, **219-220**
  modos, 15%, **291**
  pseudo-op OFFSET, **168**
  PUBLIC, **220-221**
  RET e segmentos, **96**
  superposição de segmento, **238**
  segmentos, **91**
AL, instrução LODSB, **140-141**
AND, instrução, **49-50**
Área antes do programa, **93-94, 94**
SHL, deslocamento aritmético, **55**
ASCII, códigos, **28-29, 288**
ASCII, estendidos, **290**
Asm, arquivo fonte, **68**
Assembler no Debug, **30-31**
Assembler
  automático, **126**
  comentários, **72**
  rótulos, **72**
  arquivo objeto de saída, **68-69**
  passagens, **240**
  erros de fase, **240**
  pseudo-op, **70**
  ordem de carga do segmento, **278**
  superposição de segmento, **238**
Atribuição, pseudo-op EQU, **120**
Atribuição, instrução MOV, **35**
ASSUME, **95-96, 238-239**
  grupos, **116**
  superposição de segmento, **95-96**
Atributos, inverso e normal, **160**
  IMP_ATRIBUTO_N_VEZES, **181-182**
  imprimindo caracteres e, **160**
Assembly automático, **126**
Resposta automática, LINK, **216**
AX, registrador de propósito geral, **18**

B

B, números binários, **12**
BACK_SPACE, **204**
Base 16, hexa, **7**
Base 2, binário, **12**
Modo de endereçamento relativo à base, **116**
CLS, comando do BASIC, **151-152**
Basic Input Output System, ROM BIOS, **148**
Números binários, **12, 47-48, 84**
BIOS, Sistema básico de entrada/saída, **148**
  funções na ROM de VIDEO_IO, **148**
Bits, **14**
  grupo de quatro, Nibble, **47-48**
  setando com OR, **86-87**
Block nesting error, **284**
Condições de limite, **47, 55**
Breakpoint para o G do Debug, **40**
BS, constante de retrocesso, **205-206**
Erros, descobrindo, **215, 217-218**
BX, registrador, **18**
Byte, **12**
BYTE, pseudo-op, **168-169**
BYTE PTR, **168-169**
Registradores de byte, alterando no Debug, **23**
Bytes e palavras; misturando, **168-169**

C

CALL, instrução, **57, 77-78**
  NEAR e FAR, **97**
  segmentos, **96**

a pilha, **59-60**
Flag de Carry, **36**
Controlador central, **166**
CF, Flag de Carry, **36**
erros, **109-110**
CGROUP, nome de grupo, **116-117**
Alterando a memória no Debug, **19, 22**
Alterando registradores no Debug, R, **18**
Atributos de caractere,
IMP_ATRIBUTO_N_VEZES, **181-182**
Códigos de caractere, **288**
estendidos, **290**
lendo um string, **204-205**
lendo com LE_BYTE, **168-169**
imprimindo atributos e, **160**
imprimindo strings de, **156-157**
Caracteres, **54**
CL, registro contador, **47-48, 55**
CLD, instrução, **141**
Zerando registradores com XOR, **86**
Limpando a tela, **151-152**
Limpando janelas, **151**
LIMPA_TELA, **151**
LIMPA_ATE_FINAL_LINHA, **162-163**
Interrupção do clock, **98**
CMP, instrução de comparação, **45**
comparando com 0, **86**
Segmento de código, **21**
Registrador de segmento de código, CS, **90**
COM, arquivos '.com', **70**
criando, **115**
grupos e, **116-117**
Arquivo de comando, LINK, **216**
Linha de comando, **99-100**
Comentário, **110**
Comentários e o projeto modular, **80**
Mensagens de erro comuns, **284**
Comparação com OR, **86**
Instruções de salto condicional, **44-45, 45, 47, 54-55, 62**
Constantes, CR, BS e ESC, **205-206**
Constantes, pseudo-op EQU, **120**
Convertendo binário para decimal, **84**
Convertendo decimal para hexa, **9-10**
Convertendo hexa para decimal, **7-8**
Convertendo números negativos para
complemento de dois, **14**
CONVERTE_DIG_HEXA, **196**
CR, constante de retorno de carro, **205-206**
CR, enter, **120**
CRLF, retorno de carro/mudança de linha, **119-120, 120**
CS, segmento de código, **21**
NR_SETOR, **154, 156**
Movimento de cursor, função 2 de INT 10h, **148-149, 151-153**
Cursor, leitura da posição, **160-161**
Cursor, movendo o, **151-152**
Cursor, movendo para direita, **160-161**
Cursor.asm, **120, 151, 151-152, 155-156, 160-161, 162-163**
CURSOR_DIREITA, **160-161**
CX, registrador de uso geral, **18**
CX, registrador contador, **109-110**
CY, o flag de carry, **36**

D

D, comando de cópia do Debug, **33**
Segmento de dados, **91**
múltiplo, **238**
segmento para variáveis, **116**
Tipos de dados, misturando, **168-169**
Dados
ASSUME, **238-239**
TAB_CONTROLE, **167, 215**
Modo de endereçamento imediato, **114**
superposição de segmento, **238**
DB, define byte, **97-98**
Debug, **103-104**
comando G, breakpoints, **40**
como funciona o "trace", **28-29**
comando L de carga, **102**
iniciando e saindo, **4-5**
comando T, **21, 28**
Depuração, **4-5**
PUBLIC, **106**
simbólica, **219**
técnicas, **215, 217-218**
Números decimais, convertendo de binário, **84**
Deletando caracteres. BACK_SPACE, **185**
Índice de destino, registrador de, **85**

DI, registrador, **85**
Modo de endereçamento direto, **118**
Flag de direção, **141**
Diretórios, disquete, **102**
Diretórios de disco, **102**
Setores do disco, **102**
  lendo setores com INT 25h, **130-131**
  lendo com LE_SETOR, **139-140**
  gravando, **214-215**
  gravando setores com F5, **214**
NR_DRIVE, **154, 154-155**
Es_disco.asm, **130-131, 139-140, 151, 154-158, 170, 214-215**
DISK_PATCH, **154, 157, 166-167**
Controle.asm, **166-167, 186, 188, 210, 214**
Controlador, **166, 166-167, 188, 210**
TAB_CONTROLE, **167, 215**
Cabeçalho da tela, **156**
DISP_MEIO_SETOR, **119-120, 127**
DISP_LIN, **118, 122-123, 134, 134-135, 136, 137-138**
Disp_set.asm, **114-115, 118, 121, 126-127, 134, 134-135, 136, 137-138, 139-140, 142, 152, 154, 178**
DIV, **25**
Dividindo a memória em segmentos, **19**
Divisão, **25**
  resto, **11**
Documentação, **110**
Função 25h do DOS, lendo setores, **130-131**
Funções do DOS, **294**
DS, segmento de dados, **91**
Dskpatch.asm, **153, 157**
Copiando a memória com D do Debug, **33**
DUP, pseudo-op do assembler, **92**
DX, registrador de uso geral, **18**

E

E, comando de entrada do Debug, **19, 22**
Ecoando caracteres, **61**
Editando memória, EDITA_BYTE, **188-189**
Editor.asm, **189**
EDITOR_PRONTO, **166-167**
EDITA_BYTE, **188-189**
Endereço efetivo, LEA, **129**
Fim de arquivo, **284**
END, pseudo-op, **70, 230-231**
  arquivos .EXE, **230-231**
  uso em arquivos fonte separados, **107**
Marcador de fim_de_string, **156**
ENDP, pseudo-op, **76-77, 95**
Entrando programas, **30-31**
EQU, pseudo-op, **120**
APAGA_FANTASMA, **181-182**
Apagando caracteres, BACK_SPACE, **204**
Mensagens de erro
  EXE2BIN, **280-281**
  MASM, **284**
  erros de fase, **280**
  causas possíveis, **284**
ES, segmento extra, **91**
ES, superposição de segmento, **91, 238**
ESC, constante de escape, **205-206**
OR Exclusivo, **86**
EXE, arquivos '.exe' e '.com', **91, 92**
Exe2bin, **70**
  arquivo não pode ser convertido, **280-281**
Execução passo-a-passo, **20-21**
Saída do Dskpatch — F10, **171-172**
Saída, instrução INT 20h, **29**
Códigos estendidos do teclado, **54, 290**
EXTRN, pseudo-op, **106-107**
Segmento extra, **91**

F

F1-F10, teclas de função especial, **54**
  F1 — lê setor anterior, **171-172**
  F2 — lê próximo setor, **171-172**
  F5 — grava setores modificados, **214**
  F10 — sai do dskpatch, **171-172**
CALL longo, **96**
FAR, pseudo-op, **97**
RET longo, **97**
Arquivo não pode ser convertido, **280-281**
Diretórios de arquivo, **102**
Arquivo, formato de make, **126-127**
Arquivos, nomes no Debug, **32**
Arquivos, gravando no Debug, **33**
Encontrando rotinas na memória, **217-218**
Flags

carry, **36**
direção, **141**
IRET e, **98**
estouro, **44-45**
instrução POPF, **130-131**
registrador, **98**
salvando e restaurando, **140-141**
INT e, **98**
sinal, **44-45**
zero, **44**
Discos flexíveis, **102**
diretórios, **102**
setores, **102**
lendo com INT 25h, **130**
lendo com LE_SETOR, **134-135**
gravando, **214-215**
FOR_NEXT, a instrução LOOP, **37**
Segmentos fragmentados, **281, 284-285**

G - H

G, comando do Debug, **28, 29**
breakpoints, **40**
Registradores de uso geral, **18**
GOSUB, **58**
instrução CALL, **58**
procedures, **76-77**
veja também INT, **28**
GOTO_XY, **151-152**
Caracteres gráficos, **288**
GROUP, pseudo-op, **116-117**
H, hexaritmética, **5**
números hexadecimais, **7-8**
H, para números hexadecimais
no assembler, **68-69**
Cabeçalho da tela, **156**
NR_LINHA_CAB, **154**
CAB_1, **154**
CAB_2, **154**
Hexadecimal, **6**
números no assembler, **68-69**
origens, **6**
imprimindo em, **49-50**
lendo um único dígito, **54-55**
Hexaritmética, **5**
HEXA_PARA_BYTE, **196**
Hífen, pronto do Debug, **5**

I

IF-THEN, saltos condicionais, **44-45, 45**
instrução CMP, **45**
flags de status, **46-47, 47**
Modo imediato, **118**
Instrução INC, **28, 58-59**
INC, instrução, **28, 58-59**
Incrementação, **58-59**
Índice, registradores SI e DI, **85**
Modo de endereçamento.Indireto à memória, **116**
INIC_DISP_SET, **139-140, 151, 152, 178**
Endereços de instrução, CS:IP, **90**
Ponteiro de instrução, **20-21, 21**
registrador IP, **90**
Instruções, linguagem de máquina, **19**
LEA, **129**
LODSB, **140-141**
superposição de segmento, **238**
Instrução INT, **28, 98**
INT 1, interrupção passo-a-passo, **99**
INT 10h, funções, **148, 292**
função 2, **148-149**
função 3, **149-150, 161-162, 163-164**
função 6, **149-150, 162-163, 163-164, 174**
função 9, **150, 160, 163-164**
INT 10h, Es_video em ROM BIOS, **148**
INT 20h, **29**
INT 21h, **28**
INT 21h, funções, **294**
função 1, **54-55, 56**
função 8, lendo caracteres sem eco, **61**
função 9, **33**
INT 25h, função do DOS para ler disco
setores, **128-129**
usando o comando P, **40**
Intel, **241-242**
Vetores de interrupção, **99**
Interrupção, pinha após uma, **98**
Interrupção, o clock, **98**
Interrupção, a instrução INT, **97-98**
Interrupções, retorno de, **99**
CALL intersegmento, **96**
RET intersegmento, **97**

CALL intra-segmento, **96**
RET intra-segmento, **97**
Vídeo inverso, **160**
Registrador IP, **90**
IP, ponteiro de instrução, **20-21, 21**
IRET, retorno de interrupção, **98**

J - K

JA, salto se acima, **62**
JB, salto se abaixo, **62**
JL, salto se menor que, **47**
JLE, salto se menor ou igual, **54-55**
JNZ, salto se não zero, **47**
JZ, salto se zero, **47**
Es_tec.asm, **168-169, 196, 200, 204-205, 205-206, 207-208**
Códigos estendidos do teclado, **290**
Entrada do teclado sem eco, **61**
Entrada do teclado, função 1 de INT 21h, **294**
Teclas, código de função, **54**

L

L, comando de carga do Debug, **102**
Rótulos, **72**
  endereços, **19**
  segmentos e ASSUME, **95-96**
  symdeb e, **219**
Programas grandes, **106**
  depuração, **215**
Leis do projeto modular, **108-109**
Instrução LEA, **128-129**
LET, a instrução MOV, **31**
LF, mudança de linha, **119-120**
LIFO, ordenação na pilha, **59-60**
  a pilha, **59-60**
LINHAS_ANTES_SETOR, **154**
LINK, **70**
  resposta automática, **216**
  arquivos de mapa e, **216**
  chave /map, **216**
  pseudo-ops PUBLIC, **79-80**
  ordem de carga de segmento, **122, 278**
Linkinfo, arquivo de resposta do LINK, **216**
Linkedição, **70**
  arquivos separados, **108**
  arquivos juntos, **115**
Listando um programa, comando U do Debug, **30**
Mapa de carga, **217-218**
Ordem de carga dos segmentos, **122, 278**
LODSB, instrução, **140-141**
Carregando setores, comando L do Debug, **102**
Variáveis locais, **61**
LOCATE, movimento do cursor, **151-152**
Posições de memória, endereços, **19**
Instruções lógicas, **48-49**
Operações lógicas, **86**
CALL longo, **96**
RET longo, **96**
LOOP, **37**

M

Código de máquina, **19**
Linguagem de máquina, **19, 20-21**
Make, **126**
  Formato do arquivo Make, **126-127**
Mapa, criando arquivos de, **216, 219-220**
Mapsym, criando arquivos de mapa, **219-220**
MASM
  ASSUME, pseudo-op, **240**
  mensagens de erro, **284**
  erros de fase, **240, 280**
  ordem de carga de segmento, **278**
  superposição de segmento, **238**
Memória, **19**
  endereçando instruções com CS:IP, **90**
  modos de endereçamento, **114, 291**
  pseudo-op ASSUME, **95-96, 240**
  endereçamento relativo à base, **116, 118**
  segmento de dados, **115**
  pseudo-op DB, **92**
  endereçamento direto, **118**
  dividindo em segmentos, **19**
  editando com EDITA_BYTE, **188-189**
  endereçamento indireto à memória, **116**
  rótulos para, **19**
  mapa, **216**
  deslocamento, **19**
  ordem dos segmentos na, **278**
  PUBLIC e, **220-221**

Chip de ROM, **148**
superposição de segmentos, **95-96**
registradores de segmento, **91**
segmentando, **90**
pilha na, **93-94**
Symdeb e, **219**
ALTERA_MEMÓRIA, **189**
Variáveis da memória, **153**
NR_SETOR, **155-156**
pseudo-op DB, **92**
NR_DRIVE, **154-155**
TAB_CONTROLE, **167, 215**
EDITOR_PRONTO, **166-167**
CURSOR_FANT_X, **179**
CURSOR_FANT_Y, **179**
NR_LINHA_PRONTO, **166, 166-167**
CURSOR_REAL_X, **180**
CURSOR_REAL_Y, **180**
SETOR, **121**
Microsoft e Debug, **103-104**
Microsoft, Symdeb, **219**
Misturando tipos diferentes de dados, **168-169**
Misturando palavras e bytes, **168-169**
Mnemônico, **30-31**
Setores modificados, gravando com F5, **214**
Projeto Modular, **108-109**
blocos de comentário, **108-109**
Vídeo monocromático, funções do ROM BIOS, **151**
MOV, **31**
Movendo o cursor, CURSOR_DIREITA, **160-161**
Movendo o cursor, GOTO_XY, **151-152**
MOVE_POS_ASCII, **180-181**
MOVE_POS_HEXA, **180-181**
MS-DOS e Debug, **103-104**
MUL, **24**
Segmentos múltiplos, **238**
agrupados, **116-117**
Multiplicação, **24**
por deslocamento, SHL, **55**
Multiplicando duas palavras, **24**

N - O

N, comando de nomeação do Debug, **32**
Nomes do Debug, **32**
Nomes no Symdeb, **219**
NC, flag de carry, **36**
CALL curto, **97**
Rótulos curtos, **73-74**
NEAR, pseudo-op, **76-77, 97**
RET curto, **97**
Números negativos, **13-14, 22-23**
bit de sinal, **13-14**
flag de sinal, **44-45**
PROX_SETOR, **170**
NG, flag de sinal, **44-45**
Nibble, **47-48**
Atributo normal, **160**
Números, flag de estouro, **44-45**
Números, flag de sinal, **44-45**
NV, flag de estouro, **44-45**
NZ, flag de zero, **44**
OBJ, arquivos '.obj', **70**
Arquivo objeto, saída assembler, **70**
OFFSET, pseudo-op, **168**
Deslocamento dentro do segmento, **19**
Rotinas abertas, **284**
Segmentos abertos, **284-285**
Instrução OR, **86**
CMP de um número com 0, **86**
setando bits, **86**
Ordem dos segmentos, **278**
ORG 100h, **94**
OV, flag de estouro, **44-45**
Flag de estouro, **44-45**
Estouro, o flag de carry, **36**

P - Q

P, o comando de prosseguir passo-a-passo, **40**
Fases, assembler, **240**
erros de fase, **240, 280**
Passando informações, padrões, **109-110**
Formas, IMP_FORMA, **138**
PC-DOS e o Debug, **103-104**
Fantasma.asm, **178-179, 186-187, 224, 224-225**
CURSOR_FANT_X, **179**
FANTASMA_BAIXO, **186-187**

FANTASMA_ALTO, **186-187**
FANTASMA_ESQ, **187**
FANTASMA_DIR, **187**
FANTASMA_CIMA, **186-187**
Erros de fase, **280**
  significado, **280**
PL, flag de sinal, **44-45**
POP e PUSH, **109-110**
POPF, instrução, **130-131, 141**
POS, lendo a posição do cursor, **161-162**
Números positivos, flag de estouro, **44-45**
Setor anterior, F1, **171-172**
ANT_SETOR, **170**
PRINT, função INT 21h número 9, **33**
Printaj.asm, **77-78**
Imprimindo em hexadecimal, **49-50**
IMP_A_J, **76-77**
PROC, pseudo-op, **76-77**
Rotinas, **58**
Arquivos fonte, **68**
Programas, esqueleto de, **81**
Pronto, linha de, **170**
NR_LINHA_PRONTO, **166, 166-167**
Pseudo-ops, **72**
PSP, Prefixo de Segmento de Programa, **94**
PTR, pseudo-op, **117**
PUBLIC, pseudo-op, **79-80, 106**
  arquivos de mapa e, **216**
  Symdeb e, **219**
PUSH e POP, **109-110**
PUSHF, instrução, **131**
Saindo do Dskpatch — F10, **164-165**

R

R, comando de registrador do Debug, **18**
  alterando registradores de byte, **23**
RCL, **37**
ROM, **148**
Lendo setor anterior, F1, **171-172**
Lendo próximo setor, F2, **171-172**
Lendo um string de caracteres, **204-205**
Lendo caracteres
  função 1 de INT 21h, **54-55, 56**
  LE_BYTE, **169**
  strings de caracteres, **140-141**
  sem eco, **61**
Lendo setores do disco, função INT 25h, **130**
Lendo dígitos hexadecimais, **54-55**
Lendo da memória, LODSB, **140-141**
Lendo setores
  L do Debug, **102-103**
  função 25h do DOS, **130**
  ANT_SETOR e PROX_SETOR, **170**
  LE_SETOR, **139-140**
LE_BYTE, **169, 196, 207-208**
LE_DECIMAL, **199**
  testando, **200-201**
LE_SETOR, **128, 139-140, 260**
LE_STRING, **195-196, 205-206**
CURSOR_REAL_X, **180**
CURSOR_REAL_Y, **180**
Registradores, **18**
REM, comandos de comentário, **72**
Resto, **11, 24, 25**
Removendo erros, depurando, **4-5**
Contador, instrução LOOP, **37**
RESTAURA_CURSOR_REAL, **180**
Restaurando flags, POPF, **141**
Restaurando registradores da pilha, **59-61**
RET, instrução, **58-59, 77-78**
  NEAR e FAR, **97**
  segmentos, **96**
  a pilha, **59-60**
Endereço de retorno, a pilha, **59-60**
Retorno de interrupção, IRET, **98**
RETURN, a instrução RET, **58**
Video reverso, **160**
Instrução de rotação, **37**
Rotação pelo carry, **37**
SHR, **47-48**
Rotinas no ROM BIOS, **148**

S

Saindo do Dskpatch — F10, **171-172**
SALVA_CURSOR_REAL, **180**
Salvando um arquivo do Debug para o disco, **33**
Salvando e restaurando registradores, **76, 109-110**
Salvando flags com instrução INT, **98, 99**
Salvando registradores na pilha, **59-60**
Salvando os flags, PUSHF, **141**

Código de varredura, **54**
Área de rascunho antes dos programas, **93-94, 94**
Funções da tela, **292**
Troca de tela, **219-220**
Tela, limpando a, **151-152**
Tela, usando o ROM BIOS com a, **148**
Rolamento, TELA_SOBE e TELA_DESCE, **224-225**
SETOR, **114-115, 121, 153, 154**
Apresentação de setor, INIC_DISP_SET, **139-140**
Setores, disco, **98**
DESL_SETOR, **153**
SEG, superposição de segmento, **238-239**
Deslocamento de segmentos, **19**
Superposição de segmentos, **95-96**
ASSUME e, **95-96**
instrução, **238**
erros de fase, **280**
SEGMENT, pseudo-op, **95-96**
Registradores de segmento, **91**
ASSUME, pseudo-op, **95-96**
CS, **90**
Segmentos, **19**
ENVIA_CRLF, **119-120**
Arquivos fonte separados, **106**
linkedição, **106-107, 115**
projeto modular, **108-109**
Setando bits com OR, **86**
Deslocamento, SHL, **55**
SHL, instrução deslocamento esquerdo, **55**
CALL curto, **96**
RET curto, **96**
Registrador SI, **85**
Bit de sinal, **13-14, 44-45**
Números com sinal, **11-12**
Interrupção de software, instrução INT, **97-98**
Arquivo fonte, **70**
Registrador índice fonte, **85**
SP, ponteiro de pilha, **59-60**
Teclas de função especial
entrada de teclado, **54**
lendo com LE_BYTE, **169**
tabela, **290**
Registradores de uso especial, **18**
SS, segmento de pilha, **59-60, 91**
SS:IP, **230-231**
SS:SP, topo da pilha, **92-93**
Pilha, **59-60, 230-231**
após uma instrução INT, **98**
LIFO, **59-60**
ponteiro, **59-60**
salvando flags na, **98**
segmento, **59-60**
topo da pilha, **92-93**
Padrões, **108-109**
Ponto de início em novos programas, **81**
Flags de status, **44**
Registrador de status, POPF, **130-131**
veja também Flags de status
Instruções de strings, LODSB, **140-141**
Strings, lendo, **204-205**
Strings, imprimindo com IMP_STRING, **157**
STRING_MAIUSC, **196**
SUB, **22-23**
Sub-rotinas, **58**
veja também rotinas
Subtração, **22**
instrução CMP, **45**
Trocando a tela com Symdeb, **219-220**
Chave
LINK e /map, **216**
/S do Symdeb, **219-220**
Símbolo definido mais de uma vez, **284-285**
Símbolo não definido, **284-285**
Depuração simbólica, **219**
Símbolos e PUBLIC, **220-221**
Symdeb, **219**
troca de tela, **219-220**

T - U

T, comando do Debug, **21**
Tabelas
caracteres, **288**
funções de vídeo do ROM_BIOS, **148**
modos de endereçamento, **291**
códigos estendidos do teclado, **290**
funções de INT 10h, **292**
funções de INT 21h, **294**

Armazenamento temporário, a pilha, **59-60**
TESTE, **194-195, 201**
Teste.asm, **194-195**
Limites de teste, condições de limite, **47, 55**
Testando LE_BYTE com TESTE, **194-195**
Testando LE_DECIMAL, **200-201**
Teste_seg.asm, **91**
TESTE_IMP_DECIMAL, **85, 106**
TESTE_IMP_HEXA, **79**
Texto, arquivo fonte de, **68**
As três leis do projeto modular, **108-109**
Topo da pilha, **92-93**
Procurando erros, **217-218**
Tabela verdade, AND, **49-50**
Complemento de dois, números negativos, **13-14, 22-23**
  flag de estouro, **44-45**
  flag de sinal, **44-45**
U, comando do Debug, **30**
Desassemblar, **30**
Números sem sinal, **11-12**
  JA e JB, **62**
  flag de estouro, **44-45**

V

Uso de variáveis, **110**
Variáveis e o symdeb, **219**
Variáveis
  modos de endereçamento, **114**
  FORMA_LIN_INF,
  NR_SETOR, **154, 156**
  segmento de dados, **115**
  pseudo-op DB, **92**
  pseudo-op DW, **92**
  NR_DRIVE, **154, 154-155**
  TAB_CONTROLE, **167, 215**
  EDITOR_PRONTO, **166-167**
  NR_LINHA_CAB, **154**
  CAB_1, **154**
  CAB_2, **154**
  rótulos, **72**
  LINHAS_ANTES_SETOR, **154**
  memória, **153**
  CURSOR_FANT_X, **179**
  CURSOR_FANT_Y, **179**
  NR_LINHA_PRONTO, **166, 166-167**
  CURSOR_REAL_X, **180**
  CURSOR_REAL_Y, **180**
  registradores como, **18**
  SETOR, **114-115, 121, 153, 154**
  DESL_SETOR, **153**
  superposição de segmentos, **95-96**
  FORMA_LIN_SUP,
Vetores, interrupção, **99**
Funções de vídeo de INT 10h, **148, 292**
  2, define posição cursor, **148-149**
  3, lê posição do cursor, **149-150**
  6, sobe tela, **149-150**
  9, imprime caractere/atributo, **150**
Es_video.asm, **78, 107, 114-115, 136-137, 138, 156, 156-157, 160, 163, 169, 182**

W - X - Z

W, comando de gravação do Debug, **33**
Mensagens de aviso, fontes possíveis, **284**
Aviso: sem segmento de pilha **285**
Janelas, limpando, **151**
Palavra, **12**
WORD, pseudo-op, **168-169**
WORD PTR, **168-169**
Palavras e bytes, misturando, **168-169**
Impstr.asm, **68**
IMP_ATRIBUTO_N_VEZES, **181-182**
IMP_CAR, **76-77, 77-78, 108, 119 160**
IMP_CAR_N_VEZES, **136-137**
IMP_DECIMAL, **84-85, 108**
IMP_CAB, **156, 163**
IMP_HEXA, **79**
IMP_DIG_HEXA, **79, 107**
IMP_FORMA, **138**
IMP_FANTASMA, **178**
IMP_PRONTO, **169**
GRAVA_SETOR, **214-215**
IMP_STRING, **157**
IMP_INDICES, **142**
ALTERA_MEMORIA, **189**
Gravando um arquivo com o Debug, **33**
Imprimindo um string, **33**
Imprimindo atributos, **182**
Imprimindo caracteres e atributos, **160**

Gravando setores do disco, **214-215**
Gravando setores modificados, tecla F5, **214**
Imprimindo strings de caracteres, **156**
Gravando na memória, **189**
XOR, instrução, **86**
  zerando registradores, **86**
Flag de zero, **44**
  JNZ, **45**
  JZ, **45**
ZR, flag de zero, **44**

Impressão e acabamento
(com filmes fornecidos):
EDITORA SANTUÁRIO
Fone (0125) 36-2140
APARECIDA - SP

PETER NORTON
LINGUAGEM ASSEMBLY PARA
IBM PC

Você deseja ter mais controle sobre seu PC? Mais potência? Até dez vezes a velocidade dos seus programas atuais?

Então fale com o seu PC na linguagem que ele conhece melhor: a **linguagem assembly**.

Este livro é o guia mais completo e o modo mais fácil de você dominar esta linguagem.

Você aprenderá as instruções do Microprocessador 8088 da Intel, o coração do seu PC. E, uma vez entendendo o 8088, muitos elementos vistos em outros programas e em linguagens de alto nível terão maior significado. Não apenas isso, você saberá como escrever programas completos em linguagem assembly: editores de texto, utilitários e muito mais!

Este livro pode ser usado com IBM PC, XT, AT ou compatível, com um mínimo de 128k e um drive, PC ou MS-DOS Versão 2.0 ou maior o Macro Assembler da Microsoft ou IBM.

PETER NORTON

reconhecidamente a maior autoridade mundial em microcomputadores da linha PC. Seu software *Norton Utilities* é um marco na história da computação. Seus livros *Desvendando o PC* e *PC-DOS: Como Usar o DOS com Inteligência* são os maiores best-sellers da área.

JOHN SOCHA

PhD em física aplicada pela Universidade de Cornell e projetista e autor de programas na Peter Norton Computing.

Também da EDITORA CAMPUS:
Desvendando o PC — *Peter Norton*
PC-DOS: Como Usar o DOS com Inteligência — *Peter Norton*

ISBN 85-7001-511-9